أول إصدار 7 أكتوبر 2017
الإصدار الحالي 31 أكتوبر 2022

www.code.sd

المحتويات

مقدمة

بسم الله الرحمن الرحيم، والصلاة والسلام على أشرف الأنبياء والمرسلين، نبينا محمد وعلى آله وصحبه أجمعين.

أما بعد، فإن هذا الكتاب يهدف إلى شرح أفضل الطرق لتصميم وكتابة برامج بطريقة علمية متوافقة مع مبادئ هندسة البرمجيات مما يحقق الكفاءة في إنتاج برامج وأنظمة كمبيوتر ذات جودة عالية. والكفاءة مقصود بها جهد ووقت أقل في معظم دورة تطوير البرامج من تصميم، وبرمجة، وصيانة وإضافة، واختبار، وغيرها.

كُتبت مادة هذا الكتاب بعد قراءة ودراسة من عدة مصادر في الإنترنت والتي تتحدث عن الطريقة المثلى ومفاهيم ومبادئ البرمجة الصحيحة، ثم رُبطت و أُسقطت على الواقع العملي فكانت النتيجة أن جمعت مادة الكتاب بين العملي والنظري بطريقة مبسطة وقد أُختصر بعضها لتكون كلها مفهومة ويسهل تطبيقها وليس فيها غموض.

المؤلف: معتز عبدالعظيم

تخرجت في جامعة السودان للعلوم والتكنلوجيا، كلية العلوم، قسم نظم كمبيوتر وتقنية معلومات عام 1999م. عملت كمبرمج كمبيوتر ومحلل ومعماري أنظمة ومهندس برمجيات، وقد طورت وشاركت في تطوير عدد كبير من البرامج طوال أكثر من عشرون عاماً، و ما زلت اعمل إلى الآن – بفضل الله - كمطور برامج ومعماري. وفي الآونة الأخيرة أصبحت اهتم كثيراً بالتصميم الداخلي ومعمارية الأنظمة وقد درست المبادئ الصحيحة للبرمجة، وهدفي الآن هو تطبيق تلك المفاهيم في البرامج قيد التطوير وتطلبيقها على الأنظمة القديمة.

ترخيص الكتاب

التحليل

الهدف من التحليل هو دراسة المتطلبات بصورة مفصّلة لتحديد الأهداف المطلوبة من النظام وإيضاحها للفريق المسؤول عن التنفيذ بل حتى للمستفيدين. وهو يأتي مباشرة بعد الموافقة على أي مشروع برمجي وهو البداية الحقيقية لتطوير البرامج والأنظمة المختلفة.
لابد أن يُترك للتحليل وقتاً كافياً ويقوم به ذوو الخبرة في تطوير البرامج. ولابد أن يتناسب المجهود التحليلي مع حجم المشروع البرمجي وعمره الافتراضي، فإذا كان الناتج نظام يعمل في مؤسسة يستمر معها مع استمرارها فلابد أن يكون له حيز مقدر من التحليل، فإذا افترضنا أنه لابد أن يعمل عشرة أعوام قبل تغييره أو استبداله بنظام أحدث أو حتى إعادة تصميمه، فلابد أن يكون زمن التحليل يستمر أسابيع لتغطية كل المتغيرات والميزات التي يحتاج إليها المستخدمون والنظام طوال العشر سنوات المقبلة.
في الواقع العملي يستعجل المبرمج التحليل ليبدأ مباشرة في كتابة الكود، لكن هذا الاستعجال إما أن يؤدي إلى وصول هذا المُنتج إلى نهاية عمره الافتراضي بصورة اسرع: مثلاً يعمل لعامين فقط ثم لا يتحمل التطوير والإضافات. أو يحدث أنه بعد فترة يتطلب إعادة هيكلة وإعادة تصميم لمواكبة المتغيرات التي ظهرت بعد الاستخدام والمتغيرات التي أُغفلت أثناء التحليل، والأسوأ من ذلك أن يعمل النظام ويستمر تطويره لكن بتكلفة عالية بسبب تصميمه غير الصحيح.

التصميم

التصميم هو عملية تخطيط لتحديد المواصفات الفنية والأجزاء البرمجية المختلفة التي سوف تحقق هدف النظام وعلاقاتها مع بعضها.
التصميم هو من أهم الخطوات قبل البداية في كتابة الكود، وتزداد أهميته مع زيادة حجم النظام المراد تطويره. و تُنتج مخططات تشمل هيكل النظام والمعمارية والعلاقات بين الأجزاء المختلفة من النظام.
التصميم يحتاج لمطور برامج أو مصمم أو معماري لديه خبرة في التصميم وخبرة في نوعية النظام المراد تطويره. والتصميم الجيد للنظام يضمن نجاحه و يحقق الهدف الذي بُني من أجله، ويُساهم في تقليل الجهد المبذول في التطوير والدعم الفني له.
إذا لم يُصمم النظام بصورة صحيحة أو مكتملة فسوف ينتج عن ذلك برامج غير ناجحة ومكلفة وتسبب إضاعة للموارد أثناء تنفيذ المشروع البرمجي وحتى بعد تشغليه.

كتابة الكود

تأتي مرحلة كتابة الكود عادة بعد الانتهاء من التصميم، لكن أحياناً تبدأ بعد التحليل أو أثناءه لغرض إثبات فكرة المشروع prove of concept وذلك باختيار أصعب جزء برمجي في النظام أو أهم جزء لتمثيله في شكل برنامج. مثلاً إذا كان النظام لمراقبة ماكينات تعمل في مصنع ما، فيمكن عمل برنامج ليتصل مع الماكينة ويعرض معلوماتها، مثلاً ماكينة بها منفذ RS-232 أو وصلة شبكة فيكتب المطورون برنامج مصغر بعد دراسة بروتوكول الاتصال بهذه الماكينة، الهدف من هذا البرنامج والتجارب التأكد من أن النظام يمكن عمله والهدف الفني يمكنه تنفيذه.

كذلك فإن السبب الآخر لكتابة الكود في مرحلة مبكرة هو عمل نموذج prototype وهو جزء يعمل من النظام لإثبات الفكرة وتأكيد الهدف.
بعد الانتهاء من التصميم يُقسّم المشروع بين عدد من المبرمجين والمطورين ليعمل كل واحد أو مجموعة على جزئية معينة تُربط لاحقاً لتعمل كنظام متكامل.
هذه المرحلة تأخذ الوقت الأطول في المشروع في غالب المشروعات البرمجية.

مرحلة الاختبار

يوجد عدد من أنواع الاختبار: نذكر منها اختبار الوحدات unit testing وهو اختبار لأي وحدة برمجية صغيرة أو متوسطة بطريقة مستقلة بمعزل عن باقي البرنامج أثناء أو بعد الانتهاء من أي وحدة برمجية، مثلاً إجراء أو دالة، أو نموذج إدخال مثلاً أو غيرها من المكونات المستقلة التي يمكن عمل اختبار مستقل لها. الغرض من هذا الاختبار التأكد من أن هذه الوحدة التي اكتملت تعمل بطريقة صحيحة قبل دمجها مع باقي الوحدات حتى لا ينتج خطأ أو مشكلة يصعب معرفة مكانها، حيث أن الأسهل هو اختبار الوحدات المصغرة.
النوع الثاني هو اختبار التكامل integration testing وذلك بعد ربط وحدات برمجية مع بعضها لتشكل جزئية أكبر أو تشكل النظام كاملاً، فيُختبر بتشغيل عدد من الوحدات مع بعضها للتأكد من أنها تعمل مع بعضها بالطريقة المطلوبة التي صممت من أجلها.

طريقة التصميم وطريقة البرمجة

في مرحلة تطوير المشروع أو بعد اكتماله فإن طريقة التصميم وطريقة والبرمجة تظهر للعيان في شكل نجاح المشروع أو فشله أو نجاحه إلى حد ما أو فشله إلى حد ما. ولقياس مدى نجاح أو فشل المشروع البرمجي كتبنا دلالات لقياس النجاح وأخرى تدل على الفشل في الفقرتين التاليتين:

دلالات سوء التصميم والبرمجة

سوء التصميم وسوء طريقة كتابة الكود تظهر أثناء تطوير البرنامج وينتج عنه تأخير في الوصول إلى النتائج المرجوة من المشروع. كذلك تظهر بشكل جلي بعد الانتهاء من التطوير وبداية التشغيل و تظهر مع استمرار التطوير والتعديلات. ومن أهم دلالات أن التصميم والبرمجة لم تلتزم الطريقة الصحيحة هي:

1. تأخير وتعثر في الانتهاء من الجزئيات البرمجية أو تأخير في تطوير المشروع ككل.
2. قلة الاعتمادية: أن لا يعمل النظام وفقاً لما صمم له، مثل أن يعطي نتائج غير صحيحة في حال أن المدخلات كانت صحيحة

3. التغيرات والإضافات البسيطة تأخذ وقتاً طويلاً لتنفيذها
4. التغيير في جزئية ما تنتج عنها مشكلة أو تغيير في جزئية أخرى
5. بعض المتطلبات أو التغييرات يصعب تنفيذها و أحياناً تكون مستحيلة بسبب التصميم الحالي للنظام
6. بعض التغييرات لا يمكن تنفيذها إلا بإعادة هيكلية و معمارية النظام
7. يصعب تتبع المشاكل وحلها
8. يصعب دخول مبرمجين ومطورين جدد لتطوير النظام
9. يصعب عمل اختبار جزئي unit testing
10. الاعتمادية الكبيرة للبيئة: مثل أن النظام لا يعمل إلا في نظام تشغيل معين أو نُسخة معينة منه، أو يعتمد على نظام معين، مثل اعتماده على نظام حسابات معين، فإذا حدث تبديل لهذا النظام المُعتمد عليه، تعذر ربطه مع النظام البديل.
11. عدم القدرة على تشغيل النظام في أكثر من مؤسسة: ليس به مرونة ليعمل في بيئات استخدام مختلفة ولم يُصمم ليعمل على متطلبات مختلفة.
12. الحاجة لعمل فرع/نسخة مختلفة من كود البرنامج ليُطور بمعزل عن الفرع الرئيس لخدمة أكثر من مؤسسة أو أكثر من قسم
13. عدم القدرة على تحمل زيادة الاستخدام، سواءً زيادة عدد المستخدمين أو العمليات التي تجرى عليه في الساعة، حتى مع وجود مخدمات إضافية، حيث أن بعض الأنظمة لا تستطيع الاستفادة من زيادة مخدمات جديدة ليعمل النظام عليها. عندها نقول أن النظام غير قابل للتوسع not scalable
14. وجود أجزاء برمجية في غير مكانها الصحيح: مثلاً وجود business logic في قاعدة البيانات أو في واجهة المستخدم presentation layer حيث أن مكانها الصحيح هو الجزء الوسط middle layer من هيكل النظام.
15. تعطل النظام أو إظهاره لأخطاء أو نتائج غير صحيحة أو غير مناسبة عند حدوث استثناءات عادية أثناء التشغيل، مثل انقطاع الشبكة

وعلى العكس تماماً هو التصميم الصحيح الذي يحقق الأهداف المذكورة في الفقرة التالية:

دلالات التصميم والبرمجة بطريقة صحيحة:

التصميم والبرمجة بطريقة مثالية و باستخدام مبادئ علمية ينتج عنها نظام ناجح به الميزات التالية:

1. سرعة التطوير: أن تسير مرحلة تطوير النظام بطريقة سلسة تتناسب مع حجم الأجزاء البرمجية التي طُورت
2. الاعتمادية: أن يعمل النظام وفقاً لما صمم له، ويعطي نتائج صحيحة في حال أن المدخلات كانت صحيحة
3. سهولة التعديل والإضافات في النظام
4. سرعة معرفة المشاكل وحلها
5. إمكانية إضافة أجزاء جديدة وميزات دون تغيير في هيكل النظام
6. إمكانية تشغيل نفس النظام في مؤسسات مختلفة دون تغيير في الكود الأساسي للنظام: أي يكون هُناك نسخة مصدرية واحدة فقط من كود النظام يُستخدم للعمل في كل المؤسسات المستفيدة
7. إمكانية دخول مبرمجين جدد في النظام لتطوير أجزاء جديدة أو العمل على أجزاء قديمة
8. إمكانية الاستفادة من مخدمات أخرى وذلك بتوزيع حمل التشغيل عند زيادة الاستخدام، وقابليته لتنفيذ مهام أكثر أو مستخدمين أكثر، عندها نقول أن النظام قابل للتوسع scalable
9. عمل النظام للفترة المطلوبة أو لفترة أطول من الفترة المصمم أن يعمل لها
10. قدرة النظام على التعامل مع الاستثناءات التي تحدث أثناء التشغيل بطريقة صحيحة ومتوقعة، مثل أن تظهر رسالة بأن هُناك عطل في الشبكة في حالة تعطل الشبكة أو في حالة خلل بالبيانات المُدخلة أو المخزنة.
11. سهولة عمل الاختبارات للأجزاء المختلفة من النظام

المبادئ الصحيحة للبرمجة:

توجد عدد من المفاهيم والوسائل تسمى مبادئ البرمجة programming principles تهدف لكتابة الكود بطريقة معينة وتقسيم أجزاءه المختلفة بطريقة علمية وعملية صحيحة تساهم في أن يحقق الميزات السابقة.
من هذه المبادئ:

1. التجريد abstraction
2. إعادة استخدام الكود code re-usability
3. البساطة simplicity
4. الوظيفة الواحدة single responsibility
5. إخفاء التفاصيل hide implementation details
6. الفتح والإغلاق open-closed principle
7. التماسك Cohesion
8. فك الارتباط decoupling
9. القياسية في كتابة الكود code standardization

1. مبدأ التجريد abstraction

التجريد في عالم الهندسة عموماً وهندسة البرمجيات خصوصاً من المواضيع المهمة، حيث يحقق عدة أهداف سوف نتحدث عنها بإذن الله في هذه الفقرات. لكن قبل أن نتكلم عنها من ناحية برمجية دعونا نشرحها كمفهوم عام للتجريد في حياتنا اليومية.
التجريد هو عملية التخلص من الخصائص غير الأساسية بالنسبة للهدف الذي نريد تجريده، والإبقاء على الخصائص الأساسية التي تتعلق بهذا الهدف. كذلك فهو يهدف للتعميم وليس التخصيص، وهو بذلك عكس التخصيص. لتحقيق التجريد لابد من تقسيم المشكلة، أو التصميم أو الأشياء إلى عدة مكونات أكثر عمومية و بساطة وبدائية؛ مثلاً جهاز الحاسوب إذا كان قطعة واحدة مصمتة فهو شيء معقد، إذا تلف منه جزء لابد أن يُستبدل كاملاً؛ أما التجريد -في هذا المثال- فهو يعني أن يُفصل لأجزاء مختلفة لتشكل مكونات مستقلة يمكن الاستفادة منها في عدة تطبيقات و تُركّب و تُوصل مع بعضها بطريقة اتصال معينة لتعمل في النهاية كأنها جهاز واحد. وكمقارنة بين جهاز حاسوب أكثر تجريداً وواحد آخر أقل تجريداً: أن نقارن جهاز سطح المكتب بالجهاز المحمول (لابتوب) حيث أن الأول (جهاز سطح المكتب) يتكون من أجزاء منفصلة يسهل فكها وتغييرها، مثل لوحة المفاتيح، و الفأرة، والشاشة، ووحدة المعالجة المركزية، والسماعات؛ فإذا تلف أي جزء منها، نستطيع شراء بديل له بسهولة، حيث يمكنك شراء أي لوحة مفاتيح من أي نوع ويركبها المستخدم دون الحاجة إلى مهندس حاسوب، كذلك يمكن ترقية الشاشة لأخرى ذات مساحة عرض أكبر دون أن نُغيير الحاسوب كاملاً، في هذه الحالة نقول أننا نطلب شاشة مجردة، أو مجرد شاشة؛ فبذلك يمكن استخدام أي نوع من الشاشات، أما جهاز اللابتوب فهو يتطلب شاشة مخصصة صنعت له بنفس المقاسات، لذلك نحتاج لشاشة مخصصة مناسبة له فقط. كذلك يمكنك استعارة أحد أجزاء الحاسوب المكتبي لتشغيله في حاسوب آخر، بل يمكن استخدام السماعات مع المسجل أو الراديو أو الموبايل. أما اللابتوب فهو أقل تجريداً حيث يميل لأن يكون قطعة واحدة اكثر خصوصية وأكثر تفصيلاً للاستخدام، لذلك لا تستطيع تغيير الشاشة إلى شاشة أكبر، ولا تستطيع استعارة الماوس المثبت في اللابتوب لاستخدامه مع جهاز آخر. لكن على الأقل يوجد نوع من التجريد وذلك بإمكانية ترقية الذاكرة، أوالقرص الصلب. أما الموبايل فهو أقل تجريداً من اللابتوب، حيث يميل أكثر ليكون جهاز واحد مصمت لذلك إذا تلف فإنه أصعب من ناحية الصيانة، مثلاً لا يمكن تغيير الذاكرة الداخلية إذا تلفت بسهولة، كذلك لا يمكن إعادة استخدام مكوناته مع نوع مختلف من الموبايلات. فنجد أن الأجهزة الأكثر تجريداً تتمتع بعمر خدمي أكبر، وذلك لسهولة الصيانة والترقية: وكنتيجة لذلك فإن الجهاز المكتبي لديه عمر أكبر وفرصة للصيانة أكبر من اللابتوب والذي لديه عمر أكبر ومساحة أوسع للصيانة من الموبايل، وذلك بمفهوم التجريد وليس بمفهوم جودة الصناعة، بافتراض أننا نقارن بين منتجات ذات جودة واحدة في هذا المثال.

التجريد في البرمجة هو نفس المفهوم الذي شرحناه، حيث يهدف إلى تقسيم الأنظمة الكبيرة إلى برامج أصغر ذات وظائف محددة، فبدلاً من أن يكون النظام هو عبارة عن برنامج واحد به شاشات الاستخدام، والتقارير، ويحتوي على البيانات، فإن الأفضل تقسيمه إلى طبقات: طبقة تتعامل مع المستخدم، نسميها واجهة المستخدم، وطبقة أخرى لاستقبال طلبات المستخدم وتنفيذها، وطبقة أخرى لتخزين البيانات، ويمكن تثبيت كل طبقة في مخدم منفصل حين يكون النظام كبير.

كذلك يمكن تقسيم البرامج إلى وحدات أكثر تخصصية من ناحية الوظائف، فمثلاً نظام حسابات وشيكات ومخازن، كل وظيفة يكون لها برنامج أو وحدة برمجية منفصلة يمكن تطويرها بطريقة مستقلة عن الأجزاء الأخرى. يوجد كذلك نوع من التجريد على مستوى الإجراءات وعلى مستوى كتابة الكود، فبدلاً من كتابة إجراء طويل ومعقد يمكننا تقسيم هذا الإجراء إلى عدة أجراءات أولية بسيطة، كل واحد منها يهدف إلى تحقيق وظيفة عامّة تُتيح إعادة استخدامها أكثر من مرة وفي أكثر من تطبيق.
في الأمثلة التالية عرضنا النوع الأخير من التجريد، وهو على مستوى الإجراءات.
في المثال الأول برنامج أو إجراء يُظهر معلومات المعالج في نظام لينكس، وهذه المعلومة موجودة في ملف نصي متمثل في هذا الملف و المسار:

/proc/cpuinfo

كتبنا هذا الإجراء بلغة جافا وأمسيناه *displayCPUInfo*

```
    public static void displayCPUInfo() {
        try {

            File file = new File("/proc/cpuinfo");
            BufferedReader reader = new BufferedReader(new FileReader(file));

            String line;
            while (( line = reader.readLine()) != null){
                System.out.println(line);

            }
            reader.close();
        }
        catch (Exception ex){
            System.out.println("Error: " + ex.toString());
        }
    }
```

ويكون نداءه من الدالة الرئيسة على هذا النحو:

```
    public static void main(String[] args) {
        displayCPUInfo();
    }
```

نلاحظ أن هذا الإجراء خاص، لا يمكن استخدامه لقراءة ملف نصي آخر، فقط لقراءة معلومات المعالج.
عندما نريد تجريده أولاً علينا أن نحدد ما هو الشيء الأساسي والرئيس في هذا الإجراء: وهو قراءة ملف نصي، فتكون الخطوة الثانية هي التخلص من أي تفصيل لا يُمثل هذه المهمة الأساسية. فنجد أن التفصيل الوحيد الذي لا يُمثل ولا يخدم قراءة ملف نصي عام هو اسم الملف الذي يحتوي على معلومات المعالج:

```
/proc/cpuinfo
```

فنحولّه إلى مُدخل parameter بدلاً من أن يكون ثابتاً، كذلك نغيير اسم الإجراء لأسم أكثر عمومية وهو *displayTextFile* ليتسنى استخدامه مع أي ملف.
هذا هو كود الإجراء وطريقة ندائه الجديدة:

```
    public static void main(String[] args) {

        displayTextFile("/proc/cpuinfo");
    }


    public static void displayTextFile(String filename) {
        try {

            File file = new File( filename);
            BufferedReader reader = new BufferedReader(new FileReader(file));

            String line;
            while (( line = reader.readLine()) != null){
                System.out.println(line);

            }
            reader.close();
        }
        catch (Exception ex){
            System.out.println("Error: " + ex.toString());
        }
    }
```

بهذه الطريقة يمكن استخدام هذا الإجراء الجديد لقراءة أي ملف نصي، مثلاً يمكننا قراءة وإظهار معلومات الذاكرة الموجودة في الملف :

```
/proc/meminfo
```

ثم نضيفه إلى الدالة الرئيسة:

```
    public static void main(String[] args) {
        displayTextFile("/proc/cpuinfo");
        displayTextFile("/proc/meminfo");
    }
```

بهذه الطريقة اصبح الإجراء أكثر تجريداً من السابق.

المثال الثاني لإجراء (بلغة جافا) بقراءة رسائل مستلمة لم تُعالج بعد من قاعدة بيانات ثم يضعها الإجراء في شكل مصفوفة. وفي المثال نجد أن هذا الإجراء لا يحقق التجريد، حيث أن به بعض التخصيص ويقوم بعمل أكثر من وظيفة: 1 – تجهيز الطلب لقاعدة البينات، 2 – ثم إرسال الطلب، 3- ثم قراءة السجلات ووضعها في المصفوفة. كذلك فإن هذا الإجراء مخصص فقط لقراءة الرسائل الجديدة، لا يمكننا استخدامه لقراءة رسائل قديمة مثلاً.

```
public ArrayList<SMSMessage> getNewMessages(String shortCodeText){

  success = false;
  try {
    ArrayList<SMSMessage> messages;
```

```
    PreparedStatement statement = connection.prepareStatement(
         "select * from inbox\n" +
         "where Lower(shortCode) = ? \n" +
         "and isProcessed = 0");
    statement.setString(1, shortCodeText.toLowerCase());
    ResultSet result = statement.executeQuery();
    messages = new ArrayList<>();
    while (result.next()){

      SMSMessage message = new SMSMessage();
      message.messageID = result.getInt("id");
      message.fromMDN = result.getString("From");
      message.shortMessage = result.getString("ShortMsg");
      message.shortCodeStr = result.getString("ShortCodeStr");
      message.isProcessed = result.getInt("isProcessed");
      message.shortCodeID = result.getInt("shortCodeID");
      message.msgTime; = result.getTimestamp("msgTime");
      messages.add(message);
    }

    success = true;
    return(messages);

  }
  catch (SQLException ex) {
    lastError = "Error in getNewMessages: " + ex.toString();
    General.writeEvent(lastError, "");
    return(null);
  }
 }
```

وهذا يُعد مثال بسيط، لكن توجد في الحياة العملية إجراءات أكثر تعقيداً صعبة الفهم إذا لم يُطبق مفهوم التجريد فيها.

لتحقيق التجريد سوف نُعيد صياغة الكود وذلك بقسيمه إلى جزأين: جزء خاص بطلب المعلومات وقراءة السجلات ، وإجراء آخر لوضعها في مصفوفة.

استخلصنا جزء القراءة من قاعدة البيانات وأسميناه readInbox وهو اسم مجرد يعني قراءة أي نوع من الرسائل الواردة، حيث يمكننا قراءة رسائل جديدة بها أو رسائل قديمة، بخلاف الإجراء القديم والذي لا يمكن استخدامه إلا لقراءة الرسائل الجديدة، فأصبح الكود كالتالي:

```
public ArrayList<SMSMessage> getNewMessages(String shortCodeText){

  success = false;
  try {
    ArrayList<SMSMessage> messages;
    PreparedStatement statement = connection.prepareStatement(
      "select * from inbox\n" +
      "where Lower(shortCode) = ? \n" +
      "and isProcessed = 0");
    statement.setString(1, shortCodeText.toLowerCase());

    messages = readInbox(statement);
```

```
    success = true;
    return(messages);

  }
  catch (SQLException ex) {
    String error = "Error in getNewMessages: " + ex.toString();
    System.out.println(error);
    return(null);
  }

}
```

وهذا كود الإجراء الجديد readInbox:

```
private ArrayList<SMSMessage> readInbox(PreparedStatement statement) throws
SQLException {
  ArrayList<SMSMessage> messages;
  ResultSet result = statement.executeQuery();
  messages = new ArrayList<>();
  while (result.next()){

    SMSMessage message = new SMSMessage();
    message.messageID = result.getInt("id");
    message.fromMDN = result.getString("From");
    message.shortMessage = result.getString("ShortMsg");
    message.shortCodeStr = result.getString("ShortCodeStr");
    message.isProcessed = result.getInt("isProcessed");
    message.shortCodeID = result.getInt("shortCodeID");
    message.msgTime = result.getTimestamp("msgTime");
    messages.add(message);
  }
  return messages;
}
```

يمكن إضافة إجراء جديد للحصول على رسائل قديمة:

```
public ArrayList<SMSMessage> getOldMessages(String shortCodeText, String day){

  success = false;
  try {
    ArrayList<SMSMessage> messages;
    PreparedStatement statement = connection.prepareStatement(
      "select * from inbox\n" +
      "where Lower(shortCode) = ? \n" +
      "and isProcessed = 1 and day = ?");
    statement.setString(1, shortCodeText.toLowerCase());
    statement.setString(2, day);

    messages = readInbox(statement);

    success = true;
```

```
    return(messages);

  }
  catch (SQLException ex) {
    String error = "Error in getOldMessages: " + ex.toString();
    System.out.println(error);
    return(null);
  }

}
```

نلاحظ أن الإجراء الأول (getNewMessages) أصبح أصغر وأبسط، و سهل القراءة . كذلك فإن الإجراء الجديد الذي جردناه هو (readInbox) سهل القراءة والفهم ويمكن إعادة استخدامه مع إجراءات أخرى، مثلاً لقراءة الرسائل القديمة GetOldMessages، لا نحتاج لتكرار جزئية القراءة. وبذلك نكون قد سهلنا الصيانة، مثلاً لو أضفنا حقل جديد في الجدول Inbox ما علينا إلا إضافة سطر واحد لقراءة هذا الحقل الجديد في الإجراء readInbox ، لكن إذا كان هُناك تكرار للكود ولم يكن هُناك تجريد، فإن أي تعديل أو إضافة لحقل، يتبعها تغييرات كثيرة في أي مكان من الكود يوجد فيه تكرار قراءة هذه الحقول والسجلات، وإذا نسينا تعديل جزء من الأجزاء المكررة فسوف تحدث مشكلة في البرنامج.

أهداف هندسة البرمجيات التي يحققها التجريد في الأمثلة السابقة وعموماً هي:

- تقسيم أجزاء البرنامج إلى أجزاء أولية وأبسط وأكثر عمومية
- إمكانية إعادة استخدام الكود
- منع تكرار الكود
- سهولة الصيانة لأي جزء من الكود
- سهولة القراءة والفهم ومن ثم سهولة إيجاد المشكلة
- سهولة التعديل والاختبار
- سهولة فهم البرامج من قبل باقي الفريق البرمجي

2. مبدأ إعادة الاستخدام code re-usability

تعريف إعادة استخدام الكود هو كتابة إجراء أو مجموعة إجراءات أو وحدات يمكن الاستفادة منها في أكثر من موضع. وسوف نتكلم عنها في هذه الفقرة بإذن الله من ناحية عملية.
توجد عدة طرق ومستويات لإعادة استخدام الكود، و سوف نقسمها إلى مستويات حسب مدى استخدامها، وقد قسمناها إلى ثلاث مستويات: المستوى الأول هو الأعلى قابلية للاستخدام والمستوى الثالث هو الأقل. لكن دعونا نبدأ بالمستوى الرابع:

المستوى الرابع: النسخ واللصق: وهو أن يكون لديك برنامج أو وحدة أو إجراء أو مجموعة إجراءات في نظام ما، مثلاً نظام مخازن، ونحن نريد إنشاء برنامج جديد مختلف عن نظام المخازن لكن يشترك معه في بعض الأمور، فإذا كان شديد الشبه به يمكننا نسخ نسخة من نظام المخازن ثم نستخدمها كقالب لبداية نظام آخر بدلاً من البداية من الصفر. الشكل الأخر هو وحدة في نظام ما نقوم بنسخها إلى نظام آخر ثم نعدلها لتتناسب مع النظام الآخر، وأحياناً لا تحتاج تعديل، كذلك بالنسبة للإجراءات يمكن نسخها من برنامج إلى برنامج آخر؛ بهذه الطريقة نكون قد كسبنا الوقت، فبدلاً من البداية من الصفر نبدأ بقاعدة غير قليلة من الكود. مع أننا بهذه الطريقة اعدنا استخدام كود مكتوب سابقاً إلا أننا كررناه، فاصبح نفس الكود موجود في مشروعين برمجيين في معزل عن بعضهما، أو أحياناً في نفس المشروع، والأسوأ هو أن يُطوّر و يُعدّل كل جزء من الكود بطريقة مختلفة تُناسب المشروع الذي تعمل فيه، فتصبح التعديلات في النظام الأول لا نستفيد منها في النظام الثاني مع أن مصدرهما واحد، لكن أصبحت نُسخ مختلفة من الجزء المنسوخ، مثلاً وحدة أو مكتبة في المشروع الأول كانت هي النسخة المصدر لوحدة أو مكتبة في المشروع الثاني. فإذا اكتشفنا مشكلة مثلاً أو ثغرة أمنية في هذه الوحدة فلابد أن نعدلها في جميع المشروعات التي نسخناها من هذه الوحدة، وبذلك تصبح هناك زيادة في الزمن وزيادة في التكلفة ويمكن أن ينتج عدم تطابق في أداء هذه الإجراءات، حيث أن التعديل في كل نسخة بطريقة مختلفة يجعل الإجراءات لها صفات مختلفة وبذلك فإن علاج مشكلة يمكن أن يختلف من إجراء لآخر حتى لو كان لها نفس الاسم، وتحتاج لعمل تجربة مختلفة للكود في كل مشروع. إذاً فإن طريقة النسخ واللصق هذه لا تُعد ضمن وسائل إعادة استخدام الكود بطريقة صحيحة، لأنه فعلياً توجد نُسخ مكررة مختلفة أو متشابهة في عدة مشروعات أو عدة أجزاء برمجية.

المستوى الثالث: استخدام الإجراءات: ونقصد به القيام بزيادة تجريد عدد من الإجراءات لمستوى يمكننا معه استخدامها في أكثر من موضع داخل البرنامج الواحد، فإذا كان البرنامج الناتج هو ملف تنفيذي تكون هذه الإجراءات داخل هذا الملف التنفيذي. هذه الطريقة تحقق إعادة الاستفادة من نفس الإجراء أو الوحدة داخل برنامج واحد، وتسهل صيانته حيث تمنع إعادة تكرار الكود، والتطوير لخصائص هذه الإجراءات يكون في نقطة واحدة فقط. يُستخدم هذا النوع مع الإجراءات الخاصة ببرنامج أو مشروع معين، أي أنها اكثر خصوصية بمشروع واحد وأقل عمومية ، حيث لا يمكن أن تستفيد منها باقي البرامج. مشكلة هذا النوع هو محدوديته وارتباطه بمشروع واحد، فإعادة الاستخدام طبقناها على نطاق ضيق، هو نطاق البرنامج الواحد. لكن يظل هو طريقة صحيحة يحقق مبدأ إعادة الاستخدام وننصح بتطبيقه.
هذا مثال لدالتين لإظهار المبالغ المالية بشكل محدد: بدون فاصلة عشرية للأرقام الصحيحة، وفاصلة عشرية مع خانتين للجزء الكسري إذا كان يوجد كسري في المبلغ المالي، والدالة الثانية لإظهار التاريخ والزمن بنسق معين. نحتاج لهاتين الدالتين في عدد من أجزاء البرنامج، لذلك وُضعتا في حزمة عامة للاستخدام:

```java
public static String formatMoney(double value){

       if ((int)value - value == 0){
          return String.format("%,.0f", value);
       } else {
          return String.format("%,.2f", value);
       }
   }

   public static String formatTime(Date atime){

       SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd hh:mm a");

       return format.format(atime);
   }
```

هاتين الدالتين عامتين على مستوى البرامج فقط، لا ترقى لأن تكون عامة لكل البرامج، حيث يمكن لكل برنامج أن تكون له متطلبات مختلفة من إظهار المبالغ المالية والتاريخ والوقت. لذلك وُضعتا في نفس البرنامج للاستخدام الداخلي فقط.

المستوى الثاني: الوحدات والمكتبات: وهي كتابة وحدة أو مكتبة كاملة بطريقة مجردة ليس فيها أي ارتباط بوحدات خاصة في مشروع معين ، فتُكتب بصورة عامة لتُستخدم في أي مشروع، ويوجد نوعان: النوع الأول هي الوحدات المصدرية الخاصة بلغة برمجة معينة، مثل header files في لغة سي، و units في لغة باسكال، وclass library في لغة جافا؛ أو حزمة package في لغة Go، في هذه الحال يحتاج المبرمج المستفيد من هذه الوحدة لمصدر هذا الكود حتى يضمنه المترجم مع البرنامج المستفيد أثناء الترجمة والربط، وفي حالة إنتاج برامج طبيعية Native تُضمّن هذه الوحدات داخل الملف التنفيذي فيما يُعرف بالـ static linking.
النوع الثاني: وهو استخدام مكتبة، بحيث تُترجم تلك الوحدة إلى مكتبة بصورة ثنائية حسب نظام التشغيل أو المنصة، مثلاً مكتبة .dll في نظام وندوز، ثم تُستدعى بصورة خارجية من أي برنامج، و تُربط في وقت التنفيذ ولا تُضمن ضمن الملف التنفيذي للبرنامج المستفيد، تُمسى هذه الطريقة: dynamic linking. هذا المستوى يحقق إعادة استخدام الكود بأفضل الطرق، حيث يمكن الاستمرار في تطوير تلك المكتبات بصورة منفصلة، أحياناً يكون المطور لهذه المكتبات طرف ثالث -كما يُسمى- هدفه هو فقط تطوير هذه المكتبة، وكمثال لها: مكتبات التشفير، والربط مع قواع البيانات، أو الربط مع البريد. أما مبرمجو التطبيقات فيقومون بتحميل هذه المكتبة سواءً في شكل ملف مصدري أو ثنائي لاستخدامها في برامجهم المختلف. وهي تُمثل درجة أعلى من التجريد وذلك حسب نجاحها في الاستخدام في أغراض مختلفة.

يُمكن استخدام هذه الطريقة كذلك لتقسيم المشروع الواحد إلى عدة أقسام : قسم متخصص في إجراءات البيانات، وقسم آخر للربط مع الأنظمة الأخرى، وقسم متخصص في واجهة المستخدم، وبذلك يمكن إيكال مهمة تطوير كل جزئية لمبرمج مختلف بناءً على تخصصه والمجال الذي يتميز فيه. من عيوب هذا المستوى هو الحاجة لإعادة ترجمة كل البرامج التي تستخدم تلك الوحدة في حال تغيير صفة ما في احد الإجراءات، أما إذا كان التغيير في صفة في مكتبة رُبطت خارجياً فلا تتطلب إعادة الترجمة إلا إذا تغيرت واجهة نداء الإجراء procedure interface. كذلك فإن هذه الطريقة بها ارتباط للغة البرمجة المستخدمة، مثلاً مكتبة للغة جافا لا يمكن استخدامها مع برامج سي أو برامج لغة Golang. لكن بالنسبة للغات الطبيعية مثل سي و باسكال

فيمكنها أن تتشارك في مكتبات بعضها في حال استخدام مكتبات خارجية مثل ملفات .dll في نظام وندوز، أو .so في نظام لينكس.

مثال لهذا النوع، دالة بلغة جافا للحصول على آخر يوم في شهر معين:

```
    public static int getLastDayInMonth(Date aday){

        Calendar cal = Calendar.getInstance();
        cal.setTime(aday);
        int lastDay =  cal.getActualMaximum(Calendar.DAY_OF_MONTH);
        return lastDay;
    }
```

كذلك الدالة التالية لاستخراج قيمة MD5 من نص أو مقطع مُدخل:

```
    public static String getMD5(String pass) {

        if ((pass == null) || (pass.isEmpty())){

            return "";
        }
        else {

            try {

              MessageDigest m = MessageDigest.getInstance("MD5");
              byte[] data = pass.getBytes();
              m.update(data, 0, data.length);

              BigInteger i = new BigInteger(1, m.digest());
              return (String.format("%1$032X", i).toLowerCase());
            }
            catch (NoSuchAlgorithmException ex) {
                System.out.println("Error in GetMD5: "  + ex.toString());
                return(null);
            }
       }
    }
```

يمكن استخدامها مع أي برنامج يحتاج لمعرفة قيمة MD5 لمقطع ما، مثلاً كلمة مرور.
مثل هاتين الدالتين يمكن وضعهما في مكتبة عامة للاستخدام بواسطة كل البرامج المكتوبة بلغة جافا.

المستوى الأول: الإجراءات البعيدة remote procedures، مثل خدمات الويب web services: ومن أهم أشكالها خدمات ويب يمكن نداءها من أي برنامج مكتوب بأي لغة برمجة.
و تُستخدم هذه الطريقة لربط الأنظمة مع بعضها البعض، أو لعزل جزئية من باقي الأجزاء. وتُوفّر الإجراءات بصورة حية online لأي برنامج مستفيد، كذلك يمكن تعديل الإجراءات في هذه الخدمات دون الحاجة إلى إعادة ترجمة البرامج المستفيدة، و أحياناً لا نحتاج حتى لإغلاق تلك البرامج المستفيدة عند تحديث خدمات الويب، بل في كثير من الأحيان لا نُخبر مطور البرنامج المستفيد بهذه التغييرات. تُمسى هذه الطريقة بنداء

الإجراءات البعيدة أو تُسمى أحياناً بالواجهة البرمجية API ، ومن أمثلتها خدمات الويب. و تصبح خدمة الويب هذه مورد يُصان ويُدعم من قبل جهة أخرى أو مبرمج أو فريق برمجي آخر مسؤوليته تطوير هذا الجزء المستقل من الكود. هذه الطريقة تمثل نقطة مركزية حية لكافة البرامج، فإذا حدث أي تعديل لا نحتاج لإعادة ترجمة كما في المستوى الثاني. وهو يُمثل البرمجة متعددة الطبقات، ويحقق سرية وعزل كبيرين في تطوير البرامج – وهذا موضوع آخر ليس ضمن حديثنا في هذا الكتاب- إلا أنه أقل تجريداً من المستوى الثاني، حيث أن مطور خدمات الويب لا يكون طرف ثالث في معظم الأحيان، بل هو مطور النظام الذي نريد التخاطب معه، كذلك فأن خدمات الويب لا تُمثل إجراءات بدرجة عالية من العموم تُستخدم في كل المجالات، بل هي خاصة بمؤسسة معينة تُطور خدمة الويب لها فقط، أو يمكن أن تكون خاصة بمشروع معين يُطوّر لصالح هذا النظام فقط، مثل أنظمة ERP. وبخلاف المكتبات والوحدات في المستوى الثاني فإن خدمات الويب تعتمد على مكتبات أو أجزاء كود خاصة وليست عامة.

هذا مثال لخدمة ويب عامة موجودة في الإنترنت لمعرفة رقم الإنترنت IP تابع لأي دولة، فقط نُدخل رقم الإنترنت المُراد معرفة دولته فنتحصل على معلومات الدولة، فيمكن ندائها من أي برنامج أو حتى من المتصفح:

http://services.codesoft.sd/iplocation?ip=41.209.127.216

النتيجة:


```
{
    "success": true,
    "message": "",
    "countrycode2": "SD",
    "countrycode3": "SDN",
    "countryname": "Sudan",
    "IP": "41.209.127.216"
}
```


وهذه خدمة ويب أخرى لمعرفة الوقت الحالي والمنطقة الزمنية لدولة معينة، فقط ندخل لها إختصار الدولة:

http://services.codesoft.sd/time?q=sa

النتيجة:


```
{
    "success": true,
    "errormessage": "",
    "time": "2022-05-06 08:26:18",
    "zonename": "Asia/Riyadh",
    "countryname": "Saudi Arabia",
    "gmt_offset_minutes": 180,
    "gmt_offset_hours": 3,
    "zone_abbreviation": "+03"
}
```

أو معرفة توقيت الدولة الحالية التي نُنادي منها هذا الإجراء:

http://services.codesoft.sd/time

توجد أشكال أخرى غير خدمات الويب تحقق نفس الأهداف لكنها أكثر قدما وأقل استخداماً مثل:
CORBA, RMI, COM+
هُناك ميزة مهمة في هذا النوع من إعادة الاستخدام: وهو أنه لا يوجد ارتباط أو شرط للغة برمجة معينة لتستفيد من خدمات الويب، حيث يمكن كتابة خدمة ويب باستخدام لغة جافا وندائها بلغة Go أوPHP أو أي لغة أخرى، وكذلك بالنسبة للمعماريات: حيث يمكن أن تكون خدمة ويب موجودة في منصة Solaries مثلاً يمكن أن تُنادى من منصة لينكس أو وندوز.

من ناحية عملية مع أن المستوى الثاني والأول هما من أفضل المستويات إلا أنها ليست دائماً أفضل الخيارات، فكلما ارتفع المستوى زادت التكلفة الأولية لتطوير البرامج، فلابد من اختيار المستوى المناسب مع التطبيق، فإذا كان التطبيق هو محرر نصوص بسيط فلا نتحاج لأن نطوره في شكل خدمة ويب ليُنادى من نفس جهاز المستفيد، يكفي فقط أن نستخدم المستوى الثالث والثاني كهيكلية لهذا النوع من البرامج. وكلما زاد تعقيد وحجم المشروع كانت هُناك الحاجة لاستخدام خدمات الويب لما لها من فوائد إعادة استخدام وفوائد أخرى مثل السرية وعزل البرامج من بعضها.
المستوى الأول مع أنه أكثر تكلفة في تطويره الأولي، إلا أنه يحقق أفضل النتائج لإعادة الاستخدام وصيانة البرامج وتطويرها في المستقبل. ومن ناحية عملية تُستخدم كافة هذه الأنواع في مشروع واحد.

يُستخدم المستوى الأول والثاني في واجهة البرمجة API والتي عن طريقها يمكننا ربط نظامين مع بعضهما. في السابق كان يُستخدم المستوى الثاني (استخدام المكتبات) لغرض ربط برامج مع بعضها، لكن الآن اصبح السائد هو المستوى الأول، حيث أصبحت واجهة ربط البرامج مع بعضها هو باستخدام خدمات الويب لما توفره من سرية وعزل كبير لبيئة تشغيل الأنظمة المراد ربطها ببعض.

سبحانك اللهم وبحمدك، أشهد أن لا إله إلا أنت، أستغفرك وأتوب إليك ـ

3. مبدأ البساطة simplicity

مبدأ البساطة لا يقتصر على هندسة البرمجيات فقط، إنما يشمل كل مجالات الهندسة عموماً، بكل كل مجالات الحياة. وهو من أهم مقومات المبادئ الصحيحة للبرمجة. لذلك دعونا نتكلم عن البساطة كمفهوم عام أولاً ثم نخصصها لجانب هندسة البرمجيات.
بساطة الأدوات، والتصميم، وبساطة الآلات، بل وحتى بساطة العبارات، والخُطب، والمقالات، والنظريات، وغيرها من الأشياء الملموسة وغير الملموسة في حياتنا هي من المفاهيم التي تهدف وتحقق الآتي:

1. اختيار الآلية الأبسط لتحقيق هدف ما
2. الأكثر بساطة هو الأكثر اعتماداً
3. البساطة تقلل التكلفة الابتدائية وتكلفة التعديل والتطوير وتكلفة التشغيل
4. البساطة تسّهل فهم الآخرين للأمر المراد تنفيذه ومشاركتهم في إنجازه

كانت هذه أمثلة وليست كل الأهداف والفوائد. فاختيار الطريق والآلية الأبسط لتحقيق مشروع ما أو تصليح عطل أو إنشاء بناء أو حتى كتابة كتاب فهذا لا يقلل التكلفة فقط، بل يضمن تحقيق ذلك الإنجاز. أحياناً يكون الاختيار غير الموفق للأداة – مع أنها أحياناً لا تكون جزء أساسي من المشروع- يمكن أن يساهم تعقيدها وعدم إتقانها ويمكن أن يؤدي للفشل المبكر للمشروع. فإذا كان الهدف ربط مسمار برغي واحد أو عدد محدود من البراغي فأفضل طريقة هو اختيار مفك مناسب وبسيط، فإذا اخترنا مثلاً مفك براغي كهربائي نكون قد زدنا التكلفة وزدنا التعقيد، حيث يمكن أن لا تتوفر كهرباء في المكان الذي نريد ربط البرغي فيه، فنفكر في اختيار مفك براغي كهربائي لديه بطارية أو شراء وصلة طويلة توصلنا إلى المكان الذي نريد ربط البرغي فيه، بذلك نكون قد بذلنا جهداً كبيراً في حل مشكلة الأداة (وهي مفك البراغي الكهربائي) بدلاً من الحل المباشر للمشكلة.

كذلك فإنه كلما زادت البساطة زادت الاعتمادية، حيث أن الاعتمادية تتطلب توفر الحل في جميع الظروف، مثلاً ظرف انقطاع الكهرباء، و ظرف بعد المكان عن مصدر الطاقة، وظرف العمل لوقت طويل دون توقف، وغيرها. فكلما كانت الأداة والطريقة والآلية بسيطة لتحقيق حل مشكلة ما، قل احتمال تعطلها، وبذلك زادت فرصة توفرها دائماً للعمل.
تقليل التكلفة هي شيء مهم جداً والمقصود بها التكلفة الزمنية والتكلفة المادية، حيث أنه في ظل الانفتاح وتحول العالم إلى قرية، إذا لم تقدم حلاً ذو تكلفة أقل، سوف يأتي غيرك يقدم نفس الحل بتكلفة أقل. لذلك فإن الهدف هُنا هو حل المشكلة بطريقة أكثر كفاءة على مستوى الحل الأولي وعلى مستوى الاستمرار في تشغيل وتفعيل تلك الآلية أو المشروع.
البساطة تساهم في الفهم السريع لأعضاء الفريق لآلية الحل وبذلك تسمح لهم بالتواصل و بالمساهمة الفعلية و الاستمرار و نشر فكرة المشروع بدلاً من أن يكون لغزاً محتكراً لصاحب الفكرة أو من لمن نفذه فقط.

بالنسبة لتطوير البرامج فإنها أصبحت أكثر تعقيداً من الماضي، فابتداءً بأنظمة التشغيل إلى لغات البرمجة وحتى المتصفحات، ناهيك عن عدم ذكر العتاد و ما وصل إليه من تطور، وشبكات وسرية، كلها أصبح فيها تعقيد، مثلاً المتصفحات أصبح بها مترجمات للغات برمجة مثل جافا سكريبت؛ و الاتجاه الآن هو للبساطة، بأن يكون لدينا لغات أبسط، وأنظمة أبسط ومحركات قواعد بيانات أبسط. نعني أبسط من ناحية التصميم الداخلي وحتى من حيث الإمكانات. فيمكن لأدوات ذات إمكانات أقل أن تنافس الأدوات المعقدة. وهناك

مفهوم أو نظرية في هندسة البرمجيات قرأتها مؤخراً أعجبتني كثيراً تقول worse is better، أي الأسواء هو الأفضل، ويُعنى بالأسواء هو الأقل ميزات. فإذا كُنت تريد كتابة نص بسيط فإن الأفضل هو الأدوات الأبسط مثلاً nano في نظام لينكس أو Notepad في نظام وندوز بدلاً من كتابة ذلك النص باستخدام أدوات أعقد مثل حزمة office، نكتب بها سطر أو سطرين نصيين ثم نبحث كيف نحوله إلى شكل مقروء لكل الأنظمة. أما إذا كتبناه بتلك الأدوات البسيطة فالناتج هو ملف نصي يمكن قراءته في كل أنظمة التشغيل وكل المعماريات ويمكن قراءته كذلك بجميع لغات البرمجة. تخيل أنك مبرمج و أُرسلت إليك بيانات مرجعية لاستخدامها في برنامج ما، مثلاً أرقام هواتف موظفين، فأيهما تختار: أن تكون تلك البيانات في شكل ملف word أم excel تحتاج لمكتبة لقراءتها وقراءة نسقها المعقد ثم استنباط المعلومات الأساسية منها، أم تتمنى أن يكون ملفاً نصياً بسيطاً text file خالياً من أي نسق يستطيع عمل برنامج لقراءته كل مبرمج جديد في بداية طريقة لدارسة لغة برمجة.
البساطة في تطوير البرامج يشمل اختيار الأداة الأبسط التي تستطيع إنجاز العمل وإلا أصبحت تلك الأداة عبئاً جديداً ابتداءً من تعلمها ومروراً بالبيئة التي تحتاج إليها للعمل بصورة أساسية و انتهاءً بإيجاد المبرمجين المتوفرين للعمل عليها أو للوقت الذي يحتاج إليه المبرمج لدراستها وإتقانها. كذلك فإن البساطة تشمل تصميم النظام من حيث الأجزاء المختلفة وتقنيات الربط التي يحتاج إليها، وإلى البيئة التشغيلية والتطويرية التي نطلبها لتنفيذ هذا المشرع. كذلك حتى على مستوى الكود والإجراءات فكلما استخدمنا الإجراءات القياسية البسيطة كان ذلك أفضل بدلاً من استخدام مكتبات خارجية ربما يتوقف صاحبها عن دعمها في المستقبل أو لا تدعم التطور السريع في لغة برمجة ما.
في دورة حياة أي برنامج، يبدأ بسيطاً ثم يتعقد ليلبي كافة الاحتياجات للزبائن المختلفين، فإذا زاد تعقيده من ناحية تصميمه الداخلي يشيخ ذلك البرنامج ويُحال للتعاقد لأن التطوير فيه يزيد صعوبة وبذلك تزيد ميزانية تطويره، فيكون الحل هو عمل بديل له بطريقة أبسط بتصميم نظيف clean design. فإذا بدأت بنظام بتصميم ومتطلبات معقدة فإنك بذلك تحيله إلى المعاش مبكراً. أما إذا بدأ بسيطاً وحافظ على هذه البساطة فهذا يزيد من عمره التشغيلي ويكون قليل المشاكل البرمجية ولديه فرصة أكبر ليعمل على نطاق أوسع من المستفيدين – على الأقل المستفيدين الذين يحبون البساطة-.

الأبسط هو الأفضل بالنسبة للمطور وبالنسبة للمستفيد. فالمطور يبذل جهداً و زمناً قليلين مقارنة بالأكثر تعقيداً. وبالنسبة للمستفيد الذي يبذل جهد في التعليم والتدريب للاستخدام الأمثل لهذا النظام أو البرنامج. فكما كان بسيطاً، كان الوقت اقل للتعلم، وتقل معه المتطلبات من حيث الخبرة والكفاءة للمستخدمين النهائيين الذين سوف يعملون على هذا النظام. كذلك فإن البساطة تقلل فرصة الأخطاء التي يمكن أن يرتكبوها وتسهل علاج أي خطأ يحدث.
لم ذكر أمثلة برمجية في هذه الفقرة لأن الموضوع واسع يتسع لمجالات مختلفة ولا نريد تخصيصه لجانب ونترك جانب. فكما ذكرت سابقاً فإن البساطة تشمل التصميم الداخلي للنظام، وطريقة كتابة الكود، واختيار الأدوات للتطوير، وحتى التوثيق البسيط يحقق فائدة أكبر. تخيل مثلاً وثائق للنظام بعدد أوراق معدودة مقارنة مع وثائق مفصلة ومعقدة تصل إلى مئات الصفحات، فأيهما نضمن أن المستخدم النهائي سوف يقرأه!

سبحانك اللهم وبحمدك، أشهد أن لا إله إلا أنت، أستغفرك وأتوب إليك ـ

4. مبدأ الوظيفة الواحدة single responsibility principle

فكرة هذا المبدأ ببساطة هي أن يكون للوحدة من الكود هدف واحد من كتابتها، مثلاً أن تكون وحدة متخصصة في قراءة البيانات من قاعدة البيانات العلائقية، مثلاً نسميها access unit وأخرى متخصصة في واجهة المستخدم فنسميها presentation unit، حيث تكون عبارة عن ملف أو class أو حتى طبقة كاملة Layer ، حسب التقسيم المتبع وحجم الوحدة المتخصصة. بهذا نكون قد جعلنا لكل وحدة سبباً واحداً للتغيير، مثلاً إذا كانت المهمة تغيير في واجهة المستخدم لا يؤثر ذلك على طريقة قراءة المعلومات من قاعدة البيانات، وهذا يجعل الكود أسهل تعديلاً.
نأخذ مثال لإجراء به أكثر من وظيفة ثم نصححه. والمثال مكتوب بلغة برمجة افتراضية، وهو إجراء مهمته قراءة بيانات من جدول في قاعدة بيانات ثم عرضها في المتصفح:

```
function ReadAndDisplayTable(){

  connection = Connection("localhost", "MyData", "user", "password")

  dataset = connection.query("select * from students")

  html.print("<table><tr><th>ID</th><th>Name</th></tr>")
  for record in dataset
  {
    html.print("<tr>")
    html.print("<td>", record["id"], "</td>")

    html.print("<td>", record["name"], "</td>")
    html.print("</tr></table>")
  }
  connection.close

}
```

نلاحظ أن الإجراء به قراءة من قاعدة البيانات ثم إظهارها للمتصفح، فإذا أردنا تغيير شكل المخرجات في الشاشة فإننا نُعدّل هذا الإجراء، وإذا أردنا تغيير مخدم قاعدة البيانات فنعدل نفس الإجراء، بهذا كان هُناك سببان مختلفان للتغيير، تغيير بسبب واجهة مستخدم وآخر بسبب إعدادات قاعدة بيانات، وكلاهما موضوعان مختلفان لا يمكن جمعهما في إجراء واحد، والتغيير ربما يحدث عنه خطأ غير مقصود في الجزء الآخر الذي لا نُريد تعديله من الكود، كذلك يجعل الإجراء أكثر تعقيداً وأقل قابلية لإعادة الاستخدام.

التصحيح هو أن الفصل في إجراءين منفصلين، والأفضل أن تكون وحدات مختلفة، مثلاً كل واحد في مكتبة منفصلة أو class منفصلة تجمع نوع متشابه من الوظائف:

```
function ReadTable: Dataset (){
  connection = Connection("localhost", "MyData", "user", "password")
  dataset = connection.query("select * from students")
  connection.close
  return dataset
}
```

```
function displayTable(dataset: Dataset)
{

  html.print("<table><tr><th>ID</th><th>Name</th></tr>")

  for record in dataset
  {
    html.print("<tr>")
    html.print("<td>",record["id"],"</td>")
    html.print("<td>",record["name"],"</td>")
    html.print("</tr></table>")
  }
}
```

ثم نستدعي الإجرائين من الإجراء او الدالة الرئيسة للبرنامج بهذه الطريقة:

```
data = Readtable()
displayTable(data)
```

بعد هذا التغيير يكون لدينا إجراء لقراءة البيانات فقط (readTable) وآخر لكتابته (displayTable) ونكون قد حققنا كذلك هدف إعادة استخدام الكود، حيث يمكن استخدام الإجراء readTable في أي مكان آخر ليس له علاقة بإظهار البيانات في المتصفح، مثلاً لقر ائتها ثم إرسالها بالبريد، كذلك يمكن نداء إجراء الكتابة في المتصفح بعد قراءة البيانات من ملف في القرص الصلب مثلاً.

مبدأ الوظيفة الواحدة يحقق التبسيط في كتابة الكود، فُتصبح الإجراءات والدوال أقل من حيث عدد الأسطر وتُصبح أكثر وضوحاَ، فتُصبح صيانتها وتعديلها أبسط، كذلك تُساعد في التخصص بأن يكون عدد من المبرمجيت متخصصون في واجهة المستخدم لا يتدخلون في الاجراءات المرتبطة بقاعدة البيانات والتي كتبها متخصصون في التعامل مع قاعدة البيانات.

5. مبدأ إخفاء التفاصيل hide implementation details

قبل شرح المبدأ بالنسبة للبرامج يمكننا شرحه باستخدام مثال لمطعم: حيث أن هذا المطعم يُقدم طلبات للزبائن ولديه عدد من العمال باختلاف تخصصاتهم، فلو فرضنا أن الزبون يطلب من المطعم عامل معين بإسمه أن يُجهز له طلبه، ثم من عامل آخر بتغليف الطلب ثم توصليه له، فبهذه الطريقة تكون الطلبات شخصية ومباشرة لعمال المطعم، ويمكن أن يُطلب عامل أكثر من الآخر، والعمال غير المعروفين ربما لا يتلقون طلبات للعمل، ويمكن أن يكون هذا العامل غير متوفر أو غائب، و هذا يجعل الزبائن يتدخلون في سير العمل داخل المطعم، مما يؤثر سلباً على الأداء الداخلي والدورة الخدمية للمطعم ويؤثر وعلى رضاء الزبائن. الحل هو أن تُحدد واجهة واضحة ومخصصة للتعامل مع الزبائن لطلب الوجبات، مثلاً رقم هاتف يتصل الزبون عليه ليطلب الوجبة دون أن يعتمد ذلك على عامل بعينه، أو الخدمة عن طريق نافذة بها موظف يستلم الطلبات ونافذة أخرى لتسليم الطلبات. بهذه الطريقة نكون قد أخفينا عدد من العمال الذين يعملون داخل المطعم من الوصول المباشر لهم من الزبائن، كذلك أخفينا الطريقة التي يعمل بها المطعم ومنعنا الزبائن من التدخل في سير العمل الداخلي، إنما قدمنا لهم واجهة تضمن خدمتهم بطريقة سليمة وتضمن حماية لتفاصيل العمل داخل المطعم، فهذه نفس الطريقة التي يجب أن نكتب بها البرامج: أي نُتيح للمستخدم أو المستفيد إمكانية الوصول إلى واجهة برمجية أو واجهة مستخدم دون الوصول إلى تفاصيل أبعد من ذلك في أي نظام.

بالنسبة للبرامج نريد تطبيق نفس الفكرة، أن لا نتركك للوحدات أو الفئات أو البرامج أن نصل لخصائصها وتفاصيلها من الأجزاء المستفيدة مباشرة، أي نُخفي تفاصيل التطبيق implementation، أي إخفاء الجزء الذي به المكونات الفعلية والتنفيذ الفعلي والخصائص لوحدة ما، مثل قاعدة البيانات أو الموارد المختلفة مثل الملفات و الاتصالات وحتى الإجراءات و الدوال التي تُستخدم داخلياً هي من التفاصيل التي يجب حمايتها. هذه هي مكونات وأجزاء حساسة بحيث أن تغييرها أو الوصول لها يؤثر مباشرة في سلوك النظام أو جزء منه مثل الإجراءات و الدوال الداخلية للبحث عن معلومات أو تخزينها بطريقة معينة أو حتى المتغيرات؛ حتى الأنظمة فلابد من منع أنظمة أخرى للدخول إلي تفاصيلها مباشرة مثل قاعدة البيانات الخاصة بكل نظام. لذلك يجب حماية كل تلك الوحدات من الوصول الخارجي لها ومن البيئة المحيطة بها.

الهدف من إخفاء التفاصيل هو تقليل الارتباط بين البرامج المعتمدة على جزء ما من الكود، مثلاً وحدة، أو مكتبة أو فئة، و التقليل يكون بأن لا نسمح للمستفيدين لهذه الوحدات من الوصول إلى التفاصيل مثل المتغيرات و الإجراءات الداخلية، فإذا حدث تغيير لتلك المتغيرات أو الإجراءات الداخلية، عندها لا بد من تغيير كل الوحدات المستفيدة. لكن إذا منعناها من الوصول إلى تلك البيانات والإجراءات وسمحنا لها الوصول عبر نقطة واحدة أو عدد بسيط من النقاط مثل استخدام دالة واحدة نسميها الواجهة البرمجية interface والتي تُعد المدخل الصحيح لاستخدام تلك الوحدة، أو المكتبة أو الفئة، فكلما كان عدد الـواجهات interfaces أقل وذات مُدخلات (باراميترات) أقل، كانت تلك الوحدة البرمجية أكثر استقلالية في تغييراتها الداخلية، كذلك فإن الوحدات المستفيدة تكون أقل تعديلاً إذا حدث تعديل في تلك الوحدة التي صممنا واجهة برمجية محددة لها.

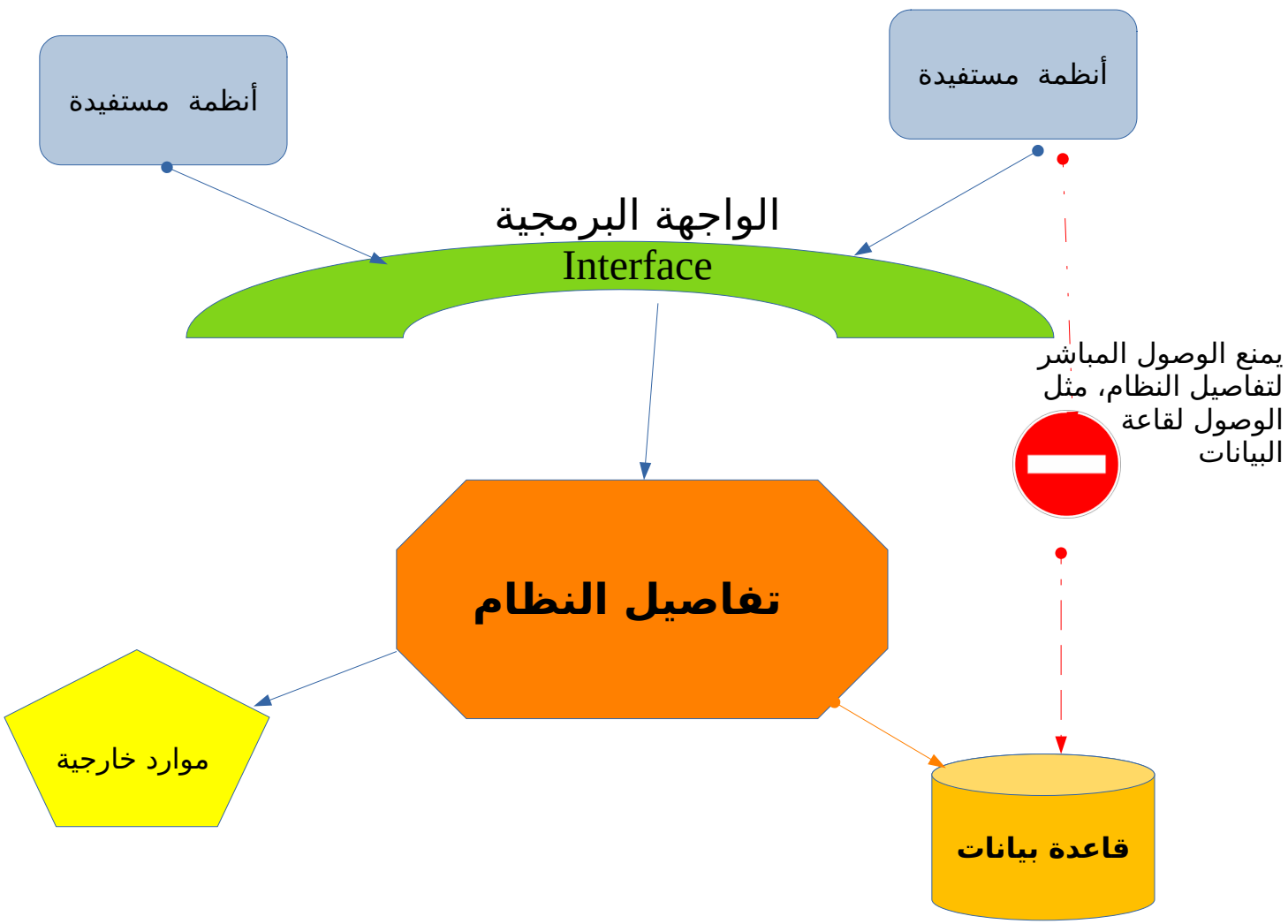
المثال التالي لوحدة unit في لغة برمجة هيكلية (لغة باسكال) لقراءة معلومات زبون من قاعدة البيانات، تفاصيل الوحدة تحتوي على الاتصال مع قاعدة البيانات وتفاصيل لغة SQL للبحث عن معلومات الزبون بالإضافة إلى متغيرات تُستخدم لتنفيذ الربط والـ query

```
unit Database;

interface

   function getCustomerInfo(customerID: string): TInfo;


implementation

var
    connection: TConnection;
    SQL: TSQLQuery;

function connectToDB();
begin
    // code for connecting to database
    connection := ....
end;

function readData(customerID: string): RecordSet;
begin
    connectoDB;
    // Query SQL
    result:= sql.Record;
end;

function getCustomerInfo(customerID: string): TInfo;
begin
   result:= readData(customerID);
end;
```

نلاحظ أننا أظهرنا دالة واحدة فقط في الواجهة interface وهي *getCustomerInfo* أما باقي الدوال الخاصة مثل *connectToDB*, *readData* والمتغيرات (SQL, connection) كلها أخفيناها بحيث لا يستطيع من يستخدم هذه الوحدة الوصول لهذه الموارد مباشرة، فقط يمكن الوصول للدالة *getCustomerInfo* بهذه الطريقة فإننا إذا غيرنا جدول قاعدة البيانات الذي نقرأ منه معلومات الزبون أو حتى لو غيرنا محرك قاعدة البيانات مثلاً فإن ذلك لا يؤثر على من يستخدم هذه الوحدة ما لم نُغيير الباراميترات وهي في هذه الحالة (*customerID*) أو اسم الدالة، ما عدا ذلك فإن أي تغيير في باقي التفاصيل المحمية بعد كلمة *implementation* فإن ذلك لا يغير في باقي الأجزاء من الكود المستفيدة من هذه الوحدة، بهذا نكون قد عزلنا المستفيدين من تفاصيل تلك الوحدة والطريقة التي تعمل بها.

نفس المفهوم يمكن تطبيقه باستخدام البرمجة الكائنية بواسطة الكبسلة *encapsulation*.

المثال السابق كان يتكلم عن وحدة صغيرة برمجية مثل الوحدات والفئات والمكتبات، لكن نفس المبدأ يمكن تنفيذه (لكن بطريق مختلف) مع الوحدات الأكبر مثل أجزاء أكبر تجمع عدة وحدات أو حتى أنظمة كاملة.
إذا تكلمنا عن أنظمة كاملة فإن إخفاء تفاصيلها له أهمية أكبر، فلا يمكن أن نسمح لنظام آخر أن يقوم بالوصول مباشرة لقاعدة بيانات نظامنا الحالي وإلا أصبحنا لا نستطيع تغيير هيكل البيانات – مثلاً - أو حتى تغيير المخدم دون أن يتأثر النظام الآخر، ناهيك عن مشاكل السرية أو مشاكل الأداء التي قد يتسبب بها النظام الآخر.
الحل الأفضل هو عمل واجهات برمجية API في شكل خدمات ويب مثلاً web services تعمل كأنها إجراءات تعمل على النظام الحالي تُنادى من أنظمة خارجية كما في الشكل أدناه :



نلاحظ أننا سمحنا للأنظمة الأخرى الاتصال بنظامنا الذي طورناه عن طريقة الواجهة البرمجة فقط API فإذا عدّلنا أي من أجزاء النظام الداخلية فإن ذلك لا يؤثر على طريقة الربط و لا يؤثر على الأنظمة الأخرى، كذلك فمن ناحية السرية والخصوصية فإن الأنظمة الأخرى لا تستطيع الوصول إلى أي تفاصيل سوى ما هو مسموح لها عن طريق الـواجهة API المحدودة والتي بدورها لا تتيح الوصول إلى كل النظام، فهي بذلك كانت مظلة و شكلت حماية لباقي النظام.

أهمية الواجهة البرمجية وواجهة المستخدم:

هذه الفقرة مهمة جداً، نرجو قراءتها بتركيز لأهميتها:

واجهة الاستخدام و الواجهة البرمجية API توفر تشغيل وظيفة معينة أو الحصول على معلومات بطريقة متفق عليها دون الدخول في تفاصيل الكيفية، حيث أن الكيفية وتفاصيلها متروكة للبرنامج أو النظام صاحب هذه الواجهة، أو متروكة للمطورين بأن يكون لهم الحرية في التصميم والتنفيذ الفني لتلك الوظائف في نظامهم، لكن عليهم الإلتزام أمام المستخدمين والمستفيدين بالواجهة وما تخدمه دون تغيير. مثلاً إذا كان هُناك نظام بنكي وبه مطورون يعملون عليه وطلبت مؤسسة ما أو بنك آخر توفير واجهة برمجية للتحويل بين الحسابات لتنفيذ عمليات شراء وبيع مثلاً أو تحويل بين حسابات بنكية، فيجب أن يكون الإتفاق على هذه الوظائف فقط: التحويل بين الحسابات، والبيع والشراء لكن ليس على تلك المؤسسة أو البنك العميل أن يتدخل في تفاصيل النظام البنكي الأول أو كيفية تنفيذ تلك الواجهة البرمجية من لغة برمجة أو المخدم الذي يحتوي على الواجهة البرمجية أو خدمة الويب أو قاعدة البيانات المستخدمة أو أي من هذه التفاصيل، فهي متروكة للمطورين أصحاب النظام الأول. وهذا موضوع مهم وإلا كان هذا سبباً لتعطيل سير تطوير النظام أو تطوير الواجهة البرمجية، وتدخل إداري وفني في التفاصيل، فلا بد أن يُعطي المطورون أهمية لهذا الموضوع، أي موضوع إخفاء التفاصيل وتوفير واجهة تعزل هذه التفاصيل الداخلية لأنظمتهم، وتوفر الوظيفة المجردة بدلاً عن التفاصيل، فالشفافية في هذه الحال تضر أكثر مما تنفع. هذه نصيحتي أشدد وأكرر على اللاتزام بها لأهميتها حسب تجربتنا.

الواجهة البرمجية لها شرطان لابد من توفرهما:

1. **الثبات وعدم التغيير**: الواجهة البرمجية هي إتفاق وهي تحتوي على بروتوكول للإتصال بين نظامين ويحتوي على وثيقة لكيفية استخدام هذه الواجهة من نداء لدوال ومُدخلات ومخرجات، فإذا حدث تغيير في النظام يجب أن تظل الواجهة البرمجية ثابتة لا تتأثر بتلك التغييرات الداخلية حتى لا يحدث إخلال بالإتفاق حول استخدام تلك الواجهة والاعتماد عليها من قبل الأنظمة الأخرى. فكما ضربنا مثال في فقرة التجريد عن خدمة ويب لمعرفة معلومات عن رقم إنترنت IP Address فمهما حدث تغيير داخلي وتحسين فلابد من الاتزام بهذه الواجهة البرمجية ومدخلاتها :
http://services.codesoft.sd/iplocation?ip=<IP-Address>

المخرجات :

```
{
    "success": <true/False>,
    "message": "<Error Message>",
    "countrycode2": "<2-Letters Country code>",
    "countrycode3": "<3-Letters Country code>",
    "countryname": "<CountryName>",
    "IP": "<IP-Address>"
}
```

إذا كان هناك اتفاق للتغيير أو إضافة جديدة فيستحسن أن تُضاف خدمة ويب جديدة أو مسار Path مختلف تحتوي على الواجهة الجديدة حتى لا تتأثر الأنظمة المتسفيدة من الواجهة القديمة.

2. **الإعتمادية**: لابد من أن تكون تلك الواجهة البرمجية يُعتمد عليها، من ناحية النتيجة المتوقعة عند النداء، ومن ناحية التوفر، أي تكون متوفرة طوال الوقت للأنظمة الأخرى لندائها وتكون مستقرة من ناحية أداء وزمة الرد – حسب سرعة الرد المتفق عليها-

إذا فشلت الواجهة البرمجية في تحقيق تلك الشروط فإن المستفيدين سوف يطلبون وصول مباشر لبعض التفاصيل كبديل لاستخدام تلك الواجهة التي لا يُعتمد عليها.

6. مبدأ الفتح والإغلاق open closed principle

احد مباديء البرمجة بطريقة صحيحة هو مبدأ الفتح والإغلاق. وهو يعني سماحية عمل إضافات في جزئية برمجية دون التعديل فيها في كود الجزئية الأساسي.
من تعريفه يبدو أن هُناك تناقض: حيث تكون هُناك إمكانية للإضافة لكن دون التعديل. والمقصود هو أن تكون لدينا جزئية برمجية أو مكتبة أو فئة كائن لديها صفات ووظائف معينة، فنعيد استخدامها مع إضافات وميزات جديدة وتوسعتها لوظائف جديدة دون أن نغير في صفاتها الأصلية ولا نغيير في مصدرها الأساسي source code. بذلك كل من يستخدم تلك المكتبة أو الكائن لا يحدث له تغيير في السلوك وذلك لأننا لم نمس تلك المكتبة أو فئة الكائن، تركناها كما هي لكن بطريقة ما اعدنا استخدامها بطريقة سمحت لنا إضافة ميزات جديدة لها مع الإبقاء على صفاتها الأصلية.
الهدف من هذه الطريقة هي إعادة الاستخدام، لكن دون المساس بالكود الأساسي كما ذكرنا، أحدى قواعدها هي إمكانية إضافة كود جديد وعدم المساس بالكود القديم. وعدم المساس بالكود الأساسي يحقق لنا استقرار البرنامج بحيث لا تتأثر الأجزاء التي تستخدم هذه المكتبة أو البرامج الأخرى التي تستخدم نفس المكتبة أو الفئة، بذلك نقلل الأخطاء التي يمكن أن تحدث بسبب تعديل أجزاء كانت مستقرة و اُختبرت من قبل وكُتبت لها وثائق لاستخدامها. ويحقق لنا توسعة قاعدة الاستخدام للكود بين البرامج.
يمكن تحقيق هذا المبدأ بعدة طرق، منها الوراثة Inheritance أو الـ Composition والتي تتطلب إمكانية برمجة كائنية للغة البرمجة المُستخدمة.

مثال للوراثة
لتحقيق هذا المبدأ باستخدام الوراثة في هذا المثال، لدينا مكتبة أو فئة للاستخدام العام في عدة برامج لتوفير دوال للتاريخ، لمعرفة السنة الحالية مثلاً والشهر واليوم، ونفترض أن هذه الدوال الثلاث تطلبها معظم البرامج أو نوعية معينة من البرامج، لذلك قررنا جمعها في فئة ومكتبة واحدة لإعادة استخدامها في عدة برامج بدلاً من إعادة كتابة الكود، كذلك لتحقيق مبدأ التجريد فهي فئة مجردة لتوفير تلك المعلومات المشتركة بين البرامج، كذلك لتحقيق مبدأ البساطة بأن كانت مكتبة بسيطة ليس فيها تعقيد أو عدد كبير من الدوال. فلذلك نريد غلقها حتى لا تنحرف عن مسارها بأن تُصبح مخصصة ومعقدة وغير بسيطة حتى تظل كما هي. لكن نريد الإضافة عليها، فنفتحها مرة أخرى لتكون أكثر فائدة وتخصيص دون المساس بها، فكيف ذلك!
الحل هو الوراثة، بأن ندع الفئة الأصلية كما هي ونرثها في فئة جديدة لنضيف دوال جديدة في الفئة الجديدة الأكثر تخصيصاً.
في المثال التالي كتبنا فئة أسميناها DateBasic وهي فئة تضم ثلاث دوال: getCurrentYear, getCurrentMonth, getCurrentDay. يمكن لأي برنامج استخدامها وذلك عن طريق الحصول على ملفات المكتبة في شكل مصدر source code أو حتى في شكل مكتبة مترجمة وثنائية JAR file لذلك في الصورة الأخيرة لايمكن التعديل عليها إذا لم يتوفر المصدر.

```
public class DateBasic {

    public  int getCurrentYear() {

        Date now = new Date();
        SimpleDateFormat formatObj = new SimpleDateFormat("yyyy");
```

```
        String year = formatObj.format(now);
        int yearInt = Integer.parseInt(year);
        return yearInt;
    }

    public int getCurrentDay() {

        Date now = new Date();
        SimpleDateFormat formatObj = new SimpleDateFormat("dd");
        String day = formatObj.format(now);
        int dayInt = Integer.parseInt(day);
        return dayInt;
    }

     public int getCurrentMonth() {

        Date now = new Date();
        SimpleDateFormat formatObj = new SimpleDateFormat("MM");
        String month = formatObj.format(now);
        int monthInt = Integer.parseInt(month);
        return monthInt;
    }


}
```

كتبنا فئة أخرى أكثر تخصصاً للاستخدام في برنامج معين لإضافة دوال متعلقة بالإسبوع مثل إسم اليوم في الإسبوع ودالة لمعرفة رقم الإسبوع في العام، ونفترض أن هذه الدوال خاصة لا تحتاج إليها كل البرامج لكن لم نكتبها في الفئة أو المكتبة الأصلية حتى لا يزيد حجمها لاحتوائها على دوال إضافية لا تستفيد منها بقية البرامج.

أسمينا الفئة الجديدة DateWeek وورثنا الفئة الأصلية DateBasic للاستفادة من دوالها الأصلية ثم الإضافة عليها:

```
public class DateWeek extends DateBasic {

    public  String getDayOfWeek() {
        Date now = new Date();
        SimpleDateFormat formatObj = new SimpleDateFormat("EEEE");
        String dow = formatObj.format(now);

        return dow;

    }

    public  int getWeekOfYear() {
        Calendar calendar = Calendar.getInstance();
        int weekOfYear = calendar.get(Calendar.WEEK_OF_YEAR);

        return weekOfYear;

    }
}
```

وهذا مثال لاستخدام الفئة الجديدة ومعها دوال الفئة الأصلية:

```
        DateWeek MyDate = new DateWeek();

        System.out.println("Current Day is: " + MyDate.getCurrentDay());
        System.out.println("Month is: " + MyDate.getCurrentMonth());
        System.out.println("Year is: " + MyDate.getCurrentYear());
        System.out.println("Day of Week is: " + MyDate.getDayOfWeek());
        System.out.println("Week of year is: " + MyDate.getWeekOfYear());
```

وهذه هي النتيجة للتشغيل:

```
Current Day is: 22
Month is: 5
Year is: 2022
Day of Week is: Sunday
Week of year is: 22
```

نجد أننا استفدنا من الفئة القديمة بوراثتها دون تغييرها، وأضفنا فئة جديدة أكثر تخصصاً في البرنامج الحالي، فأصبح هو امتداد للفئة القديمة، بذلك حققنا إمكانية تطوير تلك المكتبة أو الفئة وامتدادها إلى وظائف جديدة Extendability

مثال لتعدد الأشكال:
تعدد الأشكال Polymorphism مرتبط بالوراثة والتجريد، ومع أنه ليس الموضوع الأساسي في هذه الفقرة، لكن سوف نتحدث عنه باختصار.
تعدد الأشكال ببساطة هو أن تكون هُناك فئة كائن بها نوع من التجريد أي تصلح أن تكون عامة، مثلاً فئة كائن اسمها UserAuthenticationClass الغرض منها التأكد من المستخدم وكلمة المرور، وبها هذه الإجراءات:

```
CheckUser(String username, String password)

ChangeUserPassword(String username, String oldPassword, String newPassword)

InsertNewUser(String username, String password)
```

نحن نهدف لاستخدام هذه الفئة مع أي قاعدة بيانات لكن تختلف كل قاعدة بيانات حسب تصميمها، مثلاً أحد المبرمجين يسمي الجدول المحتوي على أسماء المستخدمين users وآخر يسميه logins وثالث يُسميه accounts. كذلك فإن الحقول التي تحتويها هذه الجداول تختلف أسمائها، كذلك فإن كلمة المرور تختلف طريقة تخزينها، فمنهم من يستخرج منها ما يُسمى بالـ hash مثل MD5 ومنهم من يستخدم شفرة لتشفير واسترجاع كلمة المرور في قاعدة البيانات، وهناك عدة طرق للتشفير أو الـ hash. لذلك يكون من الصعب أن يكون لدينا نفس الكود أو نفس المكتبة التي يمكن استخدامها مع جميع هذه الجداول المختلفة التصميم.
لحل هذه المشكلة أولاً لابد من تجريد الإجراءات الداخلية والتي تتعامل مع الجداول مباشرة، مثلاً لابد من استخلاص هذا الجزء من الكود في إجراء منفصل:

```
query.Execute("select username, password from users where username = ?");

query.setParameter(1) = username;
```

مثلاً نُسميه getAuthenticationFromTable لكن المشكلة إذا كتبنا فيه أي كود فإننا نكون قد ربطناه بتصميم معين، وهو أن يكون اسم الجدول users وحقل اسم المستخدم username وكلمة المرور password فإذا اختلف أي من هذه التفاصيل فإن هذا الكود سوف لن يعمل.
ببساطة لا نكتب متن هذا الإجراء في الفئة، فقط نكتب اسمه والمدخلات والمخرجات أي نكتب رأس الإجراء header/interface ونحوله إلى إجراء افتراضي virtual أي لا يوجد له كود implementation في هذه المكتبة أو الفئة أو الوحدة، لكن على من يستخدم هذه المكتبة أن يكتب كود هذا الإجراء بنفسه، فمثلاً هذا تعريف لهذا الإجراء في فئة الكائن الأساسي باستخدام لغة جافا:

```
public abstract ResultSet getAuthenticationFromTable(String username);
```

نلاحظ أنه لا يوجد كود بداخله، لكن مع ذلك يمكن نداءه من باقي إجراءات الفئة، مثلاً استدعيناه داخل الإجراء checkUser:

```
public boolean checkUser(String username, String password) throws SQLException{

  ResultSet result = getAuthenticationFromTable(username);

  if (result.next()){
    if (password.equals(result.getString(2))){
```

```
      return true;
    }
    else {
      return false;
    }
  }
  else {
    return false;
  }
}
```

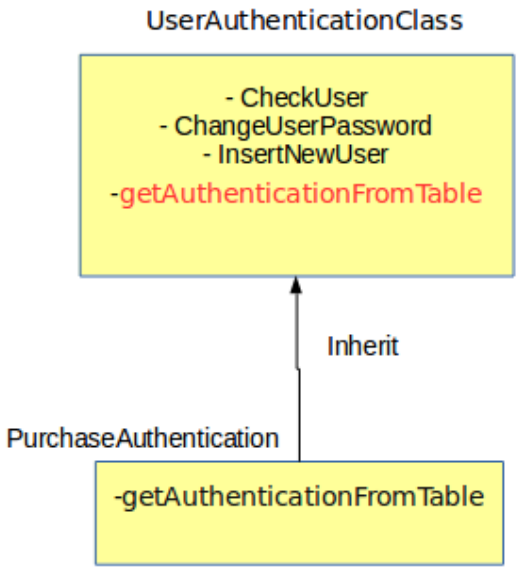
في هذه الحال لا يمكن استخدام هذه الفئة مباشرة، إنما لابد من وراثتها ثم كتابة الإجراءات الافتراضية فيها virtual، مثلاً في برنامج للمبيعات استخدمنا الوراثة في فئة جديدة أسميناها *PurchaseAuthentication* وهذا هو الكود الذي تحتويه:

```
public class PurchaseAuthentication extends UserAuthenticationClass{

  @Override
  protected ResultSet getAuthenticationFromTable(String username){

    // Do Database connection..
    PreparedStatement statement = connection.prepareStatement(
      "select username, password from users where username = ?");
    statement.setString(1, username);
    return statement.executeQuery();

  }
```

عبارة *@Override* تعني أن هذا الإجراء سوف يستبدل الإجراء الموروث، فكأنها وراثة عكسية، فبدلاً من أن ترث الفئة الابن الفئة الأب، فإن الأب في هذه الحالة يرث الابن ويستخدم إجراءه : *getAuthenticationFromTable*، لأن الأجراء في الفئة الأب لايوجد به كود implementation

نلاحظ كذلك أننا كتبنا فقط إجراء واحد لكن في المقابل اعدنا استخدام أو ورثنا كل الإجراءات الموجودة في الفئة الأصلية *UserAuthenticationClass*. عند استخدام الفئة نعّرفها وناديها كما في المثال التالي:

```
UserAuthenticationClass auth = new PurchaseAuthentication();

boolean success = auth.checkUser("mohammed", "mypassword");
```



كذلك فقد عرّفنا الكائن على أنه من الفئة *UserAuthenticationClass* لكن عندما أجرينا له تهيأة instantiation هيأناه على أنه من نوع *PurchaseAuthentication* حتى نتمكن من نداء أي إجراء من الفئة *PurchaseAuthentication*. أو من الفئة *UserAuthenticationClass*.
ويمكننا أن نقول أن الفئة *PurchaseAuthentication* هي فئة ممتدة من الفئة *UserAuthenticationClass* و الامتداد extension حققته لنا الوراثة وتعدد الأشكال بأن سمحت لنا إضافة كود جديد لكن دون أن نُغيّر الكود الأساسي للفئة الرئيسة.
ويمكننا استخدام الفئة *UserAuthenticationClass* مع أي برنامج مهما كان شكل قاعدة بياناتها وجداولها.
يمكننا كذلك تجريد الاتصال مع قاعدة البيانات ونجعلها افتراضية للسماح بإمكانية الاتصال مع أي نوع من محركات قواعد البيانات مثلا أوراكل، MySQL أو FireBird حيث أن لكل واحدة طريقة مختلفة أو مكتبة مختلفة للاتصال.
بهذه الطريقة نكون قد وسّعنا إعادة استخدام الكود بالنسبة للفئات المجردة، أو جردنا تلك الفئات ثم استخدمناها في عدد من البرامج، أي يمكننا كتابة مكتبة بها كل هذه الفئات المجردة ثم نستخدم هذه المكتبة مع باقي البرامج بدلاً من إعادة كتابة نفس الكود عدة مرات مع كل برنامج جديد مع اختلاف بسيط من برنامج لآخر. ويمكن للمبرمجين الأكثر خبرة أن يتولون مهمة كتابة هذه المكتبات حسب ما اجتمعت لديهم من الخبرة في المجال المطلوب، ثم إتاحتها لباقي المبرمجين لاستخدامها في البرامج المختلفة. بهذه الطريقة كتبنا فئة في مكتبة ثم أغلقناها لكن سمحنا لفتحها مرة أخرى عن طريق تعدد الأشكال والوراثة.

7. مبدأ التماسك Cohesion

التماسك مقصود به علاقة الإجراءات والبيانات في فئة أو وحدة واحدة. فكلما كانت الإجراءات لها علاقة وطيدة مع بعضها، كان ذلك جيداً وزاد من التماسك، أما إذا لم يكن هُناك علاقة مع بعضها أو ذات علاقة ضعيفة يُسمى ذلك قلة تماسك. من فوائد التماسك هو سهولة فهم الوحدة لصيانتها وتطويرها، ويحقق إعادة الاستخدام، وسهولة التجربة، وعند التغيير فيها لا يؤثر ذلك على باقي الوحدات.
توجد عدة أنواع أو مستويات من التماسك نبدأها بأقلها تماسكاً (أسوائها) إلى أكثرها تماسكاً (أفضلها):

أ- تماسك بالصدفة Coincidental Cohesion :
وهو وجود إجراءات ودوال في وحدة أو فئة واحدة لا علاقة لها مع بعضها، مثلاً:

```
class XYZ {

  function getMD5(string text) string {

     // calculate, and return MD5
  }

  function searchFile(string afilename) bool {

      // search a file, and return existence true/false
  }

  function downloadPage(string aurl ) string {

    // download page from URL
  }

}
```

نلاحظ في المثال السابق وجود ثلاث دوال أو إجراءات لا علاقة لها ببعضها، كذلك فهو يتعارض مع مبدأ الوظيفة الواحدة، حيث أن للفئة XYZ وظائف متعددة. فيحدث أن كل إجراء له حاجاته المختلفة وإجراءاته الثانوية المختلفة ويمكن أن لا يكون هُناك إجراءات مشتركة مما يزيد حجم الكود في هذه الوحدة البرمجية فيصعب صيانتها وفهمها، كذلك يكثر اعتمادها على وحدات أخرى مختلفة. كذلك فإن من يريد إعادة استخدام هذه الوحدة ربما يحتاج إجراء واحد فقط منها، لكن باقي الإجراءات ربما تتطلب مكتبات تمنع ترجمة البرنامج إلا بوجودها، مثلاً نفرض أن الدالة downloadPage تحتاج لمكتبة HTML والمبرمج يحتاج لهذه الفئة فقط لاستخدام الدالة searchFile فلهذا السبب لا يستطيع استخدامها إلا بوجود مكتبة HTML التي تتطلبها الدالة downloadPage مع أنه لا يحتاج إلى هذه الدالة.

ب- التماسك المنطقي Logical cohesion

ومقصود به اشتراك مجموعة من الدوال - مع اختلاف طبيعتها – تحت مسمى منطقي واحد، مثلاً فئة لتثبيت البرنامج في المخدم تحتوي على الدوال التالية:

```
class Installation {


  function writeConfig(string configData) bool {

      // write configuration file
  }

  function createDatabase(string schema) bool {

    // Create database schema
  }

  function copyApplicationFiles(string sourcedir) {

   // Copy application files into desired directory
  }


}
```

نجد أن كل إجراء ذو طبيعة مختلفة عن الآخر لكنها تشترك في مهمة تثبيت البرنامج. نجد أن كل إجراء يعمل على بيانات مختلفة.

ج- التماسك المؤقت Temporal cohesion

مثل أن يُنادى عدد من الإجراءات لا علاقة لها ببعضها عند حدث معين، مثلاً حدث مشكلة في نظام فيُرسل البرنامج رسالة نصية وبريد إلكتروني ثم يُعيد تشغيل النظام:

```
class SystemFailure {

  function sendSMS(string adminMSISDN) bool {

      // send SMS to adminsitartor(s)
  }

  function sendEmail(String supportEmail) bool {

    // send email to second line support
  }

  function restartSystem() {

   // Restart for application server

  }
}
```

نلاحظ أنها إجراءات جمع بينها حدوث حدث معين ومؤقت. وهي ذات طبيعة مختلفة ومُدخلات مختلفة.

د- التماسك الإجرائي procedural cohesion:
وهي دوال تُجمع في دالة واحدة بسبب أنه سوف تُنادى بالتتالي لإتمام مهمة واحدة مع أن كل إجراء يختلف في طبيعته. مثلاً إجراء لضغط ملف ثم تشفيره ثم إرساله كما في المثال التالي:

```
class PrepareAndSendFile {

  function CompressFile(string source, string dest) string {

     // Compress file
  }

  function EncryptFile(string source, string dest, string key) string {

      // Encrypt file
  }

  function sendFile(string afilename, string toaddress ) bool {

    // send file to email address
  }

}
```

نلاحظ أن كل دالة لها طبيعة مختلفة وتحتاج لمكتبات مختلفة، لكن ما جمعها أنها تُنادى بالتتالي. بها مشكلة الطبيعة المختلفة والبيانات أو المدخلات المختلفة.

هـ- التماسك المعلوماتي أو الاتصالي Communicational/informational cohesion:
والمقصود به وجود دوال مع بعضها لأنها تشترك في معالجة نفس البيانات مثلاً لمعالجة سجل أو مجموعة سجلات، أو جدول، كما هذا المثال:

```
class UsersData {

  function insertNewUser(string username, string password) bool {

     // insert new user into table
  }

  function modifyUserPassword(string username, string newpassword) bool {

      // modify user password
  }

  function getUserInfo(string username) record {
```

```
    // Get user record
  }

}
```

و- التماسك المتتالي Sequential Cohesion

والمقصود به أن تكون مخرجات دالة تُستخدم كمدخلات لدالة أخرى كما في هذا المثال:

```
class Users {

  function getUsers() recordset {

     // fetch table and return recordset
  }

  function searchUser(string username, recordset records) record {

      // search user in recordset
  }

  function uploadUserInfo(record user, string aurl ) bool {

    // send user info into a web service
  }

}
```

حيث أن الإجراء الأول getUsers يسترجع كافة السجلات للمستخدمين ثم تُستخدم هذه السجلات كمُدخلات للدالة الثانية searchUser والتي تبحث عن سجل واحد لمستخدم ، ثم ترسله الدالة الثالثة uploadUserInfo إلى خدمة ويب.

ز- التماسك الوظيفي Functional cohesion

وهو من أفضل أنواع التماسك والمقصود به أن تشترك دوال وإجراءات لتحقيق هدف واحد وواضح ومحدد.
مثلاً وحدة لضغط عدد من الملفات:

```
class Compression {

  function AddFile(string afilename) bool {

     // Add file to prepare it for compression
  }

  function compress(string archivename) bool {

      // Compress all added files
  }
```

```
  function getCompressionSize() int {

    // Get compressed archive file size
  }

}
```

نجد أنها كلها تخدم وظيفة واحدة وهي وظيفة ضغط الملفات.

ح- التماسك الذري Perfect cohesion (atomic)

وهو أن تكون الوحدة تحتوي على كود يمثل إجراء واحد لا يمكن تقسيمه إلى وحدة أصغر، مثلاً:

```
class Encryption {

  function encryptFile(string archivename, string key) string {

      // encrypt file
  }

}
```

الأنواع التي يمكن استخدامها وتمثل المبادئ الصحيحة لطرق البرمجة هي:

- هـ) التماسك المعلوماتي
- و) التماسك المتتالي
- ز) التماسك الوظيفي (أفضلها)
- ح) التماسك الذري (عملياً يصعب تطبيقه في المهام المعقدة)

8. مبدأ فك الارتباط decoupling

قبل الكلام عن فك أو تقليل الارتباط نتكلم عن ماهو المقصود بالارتباط: الارتباط هو ارتباط وحدتين أو إجراءين بحيث لا يمكن استخدام إجراء أو وحدة أو فئة إلا باستخدام الأخرى، أو أن عمل وحدة ما تؤثر على الأخرى، أي أن الوحدات البرمجية غير قائمة بذاتها وغير متماسكة. وهذا من التصميم غير الصحيح في البرمجة، كذلك فإنه يجعل الوحدات المرتبطة غير قابلة لإعادة الاستخدام وصعبة في التطوير وصعبة في اكتشاف الثغرات وصعبة في إجراء الاختبارات.
المثال التالي به فئة مرتبطة بأخرى ولا تعمل بدونها:

```
class SendEmail {


  public static void sendEmail(String to, String text){

     // sending Email.text to Email.address

  }
}

class Alarm {

  public void SendAlarm(String to){

    SendeEmail.sendEmail(to, "System has failed");

  }
}
```

نجد أن الفئة Alarm مرتبطة ارتباط وثيق بالفئة SendEmail، وسوف لن تعمل بدونها.
مثلا لا نستطيع أن نستبدل الإرسال بالبريد برسالة قصيرة عبر الموبايل مثلاً، إلا إذا فصلنا الارتباط بفئة الإرسال عبر البريد واستبدلناها بفئة عامة للإرسال يمكن استخدامها مع أي نوم من وسائل الاتصال، فعرّفنا فئة مجردة اسمها Sender لا تحتوي على آلية إرسال حقيقي إنما فقط واجهة إجراء يُسمى send:

```
public abstract class Sender {

    abstract void send(String to, String text);

}
```

ثم كبتنا فئتين ترثان هذه الفئة إحداهما للإرسال عبر البريد وأخرى للإرسال عبر الرسائل القصيرة، وبهما آلية الإرسال الفعلية:

```
public class SendEmail extends Sender {

    @Override
    void send(String to, String text) {
        // .. Sending by Email
```

```
    }
}

public class SendSMS extends Sender {

    @Override
    void send(String to, String text) {

        //  Sending SMS
    }

}
```

ثم تعريف الفئة Alarm بالطريقة التالية الخالية من الارتباط بنوع الإرسال سواءً بريد أو رسائل قصيرة:

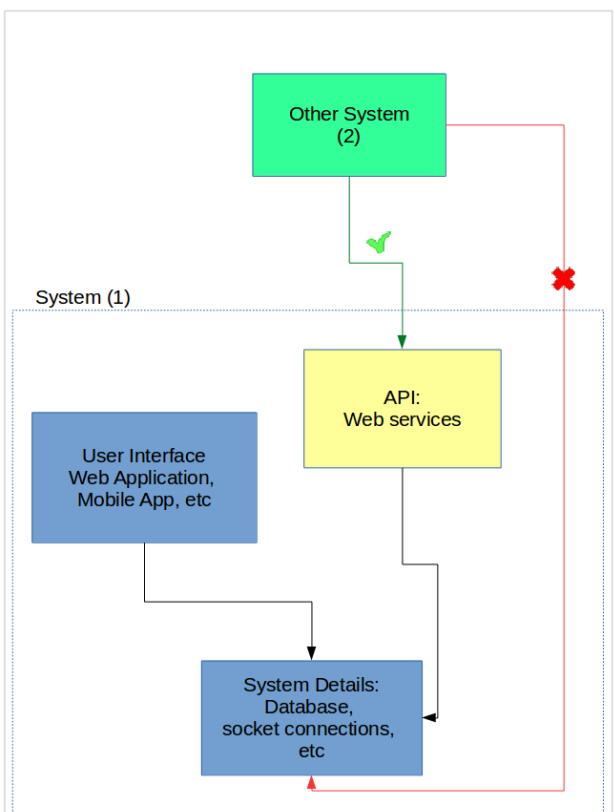
```
public class Alarm {

    public void SendAlarm(Sender sender, String to){

        sender.send(to, "Alarm text");

    }

}
```

نلاحظ أننا وضعنا الفئة العامة Sender كمُدخلات تُوضع عند طلب تنفيذ إجراء CaptureAlarm، ففي هذه الحالة لم تُربط الفئة Alarm بالفئة: SendEmail أو SendSMS إنما أدخلنا الفئة المجردة كوسيط وهي Sender لفك ذلك الإرتباط الوثيق، أي قللنا الأرتباط أو ما يُسمى loosely coupling

وعند مناداتها نختار الطريقة التي نريد بها الإرسال ثم إرسال الفئة المحتوية على طريقة الإرسال إلى الإجراء SendAlarm:

```
    public static void main(String[] args) {
        // TODO code application logic here

        Alarm alarm = new Alarm();
        Sender send = new SendEmail();
        alarm.SendAlarm(send, "test@test.com");
    }
```

هذا مثال في مجال كتابة الكود لتقليل الارتباط بين دالتين أو فئتين، وكمثال في مجال لتصميم البرامج نجد هناك عدة أشكال من الارتباط ومن أشهرها الارتباط ببيانات في جزئية خارجية مثل قاعدة البيانات، مثلاً النظام 2 مرتبط بالنظام 1 في قاعدة البيانات الخاصة بالنظام 1، فعند تغيير معلومات في جدول ما أو تغيير في هيكله فإن ذلك يؤثر على النظام 2، وهذه من الأخطاء التصميمية الكبيرة، تجعل تعديل النظام 1 صعباً وذلك لأن أي تغيير في قاعدة بيانات يؤثر على أنظمة أخرى ليست جزءاً من النظام الأول.
لحل مشكلة الارتباط في قاعدة البيانات تُطور خدمات ويب في النظام 1 ثم تُتاح لباقي الأنظمة، فإذا حدث تغيير في هيكل قاعدة البيانات الخاصة بالنظام 1 يغير المطورون في الكود الداخلي لخدمة الويب الخاصة به فقط ولا يغيروا واجهتها البرمجية، لذلك لا تتأثر باقي الأنظمة بهذا التغيير وأحياناً لا تحتاج حتى لمعرفة أن هُناك تغيير حدث في قاعدة البيانات. كذلك يمكن تغيير في معمارية النظام 1 دون أن يتأثر النظام 2 بهذا التغيير، كما في الصورة أدناه:



سبحانك اللهم وبحمدك، أشهد أن لا إله إلا أنت، أستغفرك وأتوب إليك ـ

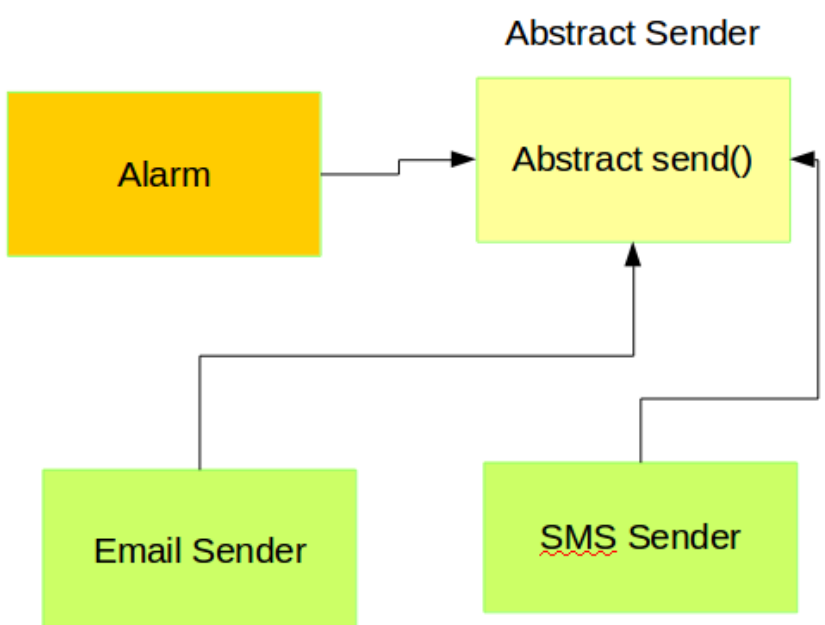
طريقة Dependency Inversion

وهي من طرق فك الارتباط وتنص على القاعدتين التاليتين:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

القاعدة الأولى تعني أن الأجزاء أو الوحدات الأعلى أي الأقرب لواجهة المستخدم يجب أن لا تعتمد على الأجزاء الدنيا مثلاً التي تتصل بقاعدة البيانات، والسبب أننا لو اضطررنا إلى تغيير تلك التفاصيل أو طريقة الاتصال بقاعدة البيانات أو تغيير قاعدة البيانات مثلاً سوف نضطر لعمل تغيير في الأجزاء العليا من الكود. بدلاً من ذلك يجب على الأجزاء العليا والدنيا الاعتماد على التجريد، أي أن التجريد هو ما يُرتبط به.

والقاعدة الثانية: تعني أن التجريد يجب أن لا يعتمد على التفاصيل، بل التفاصيل هي من تعتمد على التجريد.
ولشرح هذا المبدأ نرجع للمثال السابق في الصورة التوضيحية أدناه:
نجد أن الفئة Alarm هي فئة عليا أقرب للمستخدم، والفئات EmailSender, SMSSender, هي فئات دنيا، لذلك لا يجب أن تعتمد الفئة العليا Alarm على أي واحدة منها، بدلاً من ذلك يجب أن تعتمد على الفئة المجردة Sender والتي تعتمد عليها الفئات الدنيا EmailSender, SMSSender.
بهذه الطريقة يمكننا إضافة أي نوع إرسال جديد كفئة دنيا دون أن نقوم بعمل تغيير في الفئة العليا Alarm.



طريقة حقن الارتباط Dependency Injection

الآلية التي استخدمناها لنداء دالة الإرسال في الدالة الرئيسة main تُسمى بحقن الارتباط Dependency Injection، حيث عرّفنا النوع الذي نريد استخدامه في الإرسال وهو SendMail ثم أرسلناه كباراميتر لفئة Alarm كالتالي:

```
        Alarm alarm = new Alarm();
        Sender send = new SendEmail();
        alarm.SendAlarm(send, "test@test.com");
```

9. مبدأ القياسية في كتابة الكود code standardization

مقصود به استخدام العرف القياسي في طريقة كتابة الكود سواءً للغة البرمجة المستخدمة، أو نوعية البرامج التي تطوّر.
استخدام القياسية في كتابة الكود يحقق هدف كتابة كود سهل القراءة والصيانة والفهم. يمكن أن تكون لكل لغة برمجة طريقة قياسية في كتابة الكود، أو يمكن أن يكون لكل مؤسسة برمجية عرف قياسي convention ومنهج محدد لكتابة وتصميم البرامج، لكن الأفضل أن يكون مأخوذ من عرف قياسي عام، ثُم يمكن التعديل عليه ليناسب المؤسسة ونوعية مشروعاتها البرمجية، واتباع هذا العرف يُساعد على إدارة المشروعات البرمجية وصيانتها بطريقة سهلة.
وكمثال ولشرح أوفى لهذا المفهوم هو موضوع التسميات في الأجزاء البرمجية مثل المتغيرات، والإجراءات، و الكائنات، والوحدات وحتى البرامج وأجزاءها المختلفة.
التسمية الصحيحة والمعبرة والدقيقة لكل جزئية من البرامج لا تقتصر فائدتها على كتابة كود مقروء فقط، إنما تتعدى ذلك لفوائد أخرى لا تقل أهمية، وهي أنها مقياس لصحة التصميم الداخلي للبرامج، ومقياس لمهنية المبرمج وفهمه الصحيح لهذه الطرق و المبادئ للتصميم وكتابة البرامج، بل وحتى لها دلالات لفهم الفريق البرمجي للمشكلة التي من أجلها طُور البرنامج أو النظام.
تسمية الإجراءات على سبيل المثال والكائنات لابد أن يكون شاملاً ومختصراً ويعبر عن الوحدة البرمجية المستهدفة بطريقة مباشرة لوظيفتها. مثلاً إذا كان المراد كتابة دالة لقراءة معلومة من ملف إعدادات يمكن أن يكون اسمها كأحد الخيارات التالية:
readConfigValue, *getConfigValue*, *readConfigParameter*, *getConfigurationParameter*
نلاحظ أنها واضحة وتتكون من ثلاث مقاطع، المقطع الأول *read* يعني أن هذه الدالة تقوم بالقراءة، والمقطع الثاني: *Config* يشرح الهدف الذي نريد قراءته، والمقطع الثالث *Value* يحدد بصورة أكثر دقة الناتج من الدالة وهو قراءة القيمة لهذا البارامیتر.

أما إذا استخدمنا احد الأسماء التالية:
configValue, *readX*, *getValue*
نجد أن الاسم الأول: *configValue* ليس فيه الفعل الذي نريد تنفيذه وهو القراءة، لذلك يجد من يريد صيانة الكود أو تعديله، يجد صعوبة فهم الغاية من هذه الدالة، و الاسم الثاني *readX* لا يوضح ماذا نريد أن نقرأ، و الاسم الثالث: *getValue* كذلك لا يوضح أننا نريد القراءة من ملف الإعدادات.

كذلك فإن الشمولية في الاسم مهمة جداً، ويُفضّل أن تكون مجردة قدر الإمكان، فإذا كانت الفئة أو الوحدة مثلاً المراد منها القراءة والكتابة من قاعدة البيانات فلا يصح أن نقوم بتسميتها *DataReader* فهي لا تشمل الكتابة فالأفضل تسميتها *DataReaderAndWriter* أو الاكتفاء بالتسمية المجردة وهي *DataClass* أو *DataManipulation* أو *Database*. والتسمية المجردة لها فائدة في التطوير المستقبلي، مثلاً إذا كانت الدوال داخل تلك الوحدة البرمجية هي فقط دوال للقراءة فإن تسميتها *DataReader* يكون صحيحاً ما لم تُضاف دالة للكتابة، أما إذا أُضيفت وظيفة الكتابة في قاعدة البيانات فلابد من تغيير الاسم، لكن إذا نسى المبرمج ذلك فإن ذلك يقود إلى أن تصبح الأسماء غير مطابقة للوظائف ويبدأ البرنامج بالحيد من إتباعه للمبادئ الصحيحة للبرمجة، ويصبح صعب الفهم.

في آخر برنامج كنت اعمل عليه في الأيام الماضية وجدت أني أتوقف كثيراً في تسمية الفئات وتصنيفاتها المختلفة وكان الهدف أن لا أقوم بعمل أي خطأ في بداية التصميم وكتابة الكود، لكن هذا التأخير في وجود الأسماء المناسبة جعلني أفكر في أن السبب هو عدم فهمي الكامل للتحليل الصحيح للنظام. فاستنتجت أن التحليل الكامل ثم الفهم التام للنظام يجعل التصميم أسرع و إيجاد أسماء للفئات بطريقة أسرع. مع أني لم أتأكد من هذه النظرية من مرجع ما، إلا أن هذا الارتباط بين وضوح التحليل والتصميم الصحيح والتسميات اصبح يتضح لي أكثر فأكثر، حيث يكفي مراجعة الأسماء لأي برنامج لمعرفة هل هُناك خلل تصميمي ما أم أن البرنامج مكتوب بطريقة مطابقة للمبادئ الصحيحة. هذه الفائدة لربط الأسماء بالمبادئ تسهل لمن يقوم بمراجعة الكود من أجل تقيمه، فكلما كانت الأسماء مجردة فهي تعني أن المبرمج أو الفريق البرمجي قام باستخدام مبدأ التجريد، وكلما كانت الأسماء تدل على هدف واحد فهي تعني استخدام مبدأ الوظيفة الواحدة، فإذا كان الاسم يحتوي على أكثر من هدف مثل *DatabaseAndConfiguration* فهي تدل على تحميل الوحدة لأكثر من هدف، أما إذا كان الاسم هو *Database* ويحتوي كذلك على قراءة للإعدادات من الملفات فهو يعني عدم شمولية الأسماء، لكن الاسم الشامل في هذه الحال يدل على مشكلة أخرى و هي عدم استخدام مبدأ الوظيفة الواحدة، لكن مع ذلك فإن الاسم الشامل أفضل في هذه الحالة لمن يريد تعديل الكود وفصل المهام عن بعضها، فيُركز على الوحدات التي تحتوي على أسماء لأكثر من وظيفة لفصلها في أكثر من فئة.

تعديل الأسماء أحياناً يكون سهل وذلك في الأجزاء الداخلية للبرامج، فتُعدّل الأسماء لعدة حالات نذكر منها: الحالة الأولى إذا لم تكن دلالاتها صحيحة، في هذه الحال تكون ضمن عملية إعادة صياغة الكود، وذلك لتصحيح الهدف من تلك الوحدة البرمجية. والسبب الثاني هو توسعة الوظيفة، وكمثال ما ذكرناه سابقاً في الوحدة التي كان اسمها *DataReader* جردنا اسمها إلى *DataClass* لإضافة وظيفة الكتابة مع القراءة في قاعدة البيانات. السبب الثالث هو التجريد لإعادة الاستخدام في برامج أخرى أو أجزاء أخرى. مثلاً إجراء للربط مع قاعدة البيانات المحاسبية كان اسمه *AccountingDBConnection* نقوم بتسميته إلى *DatabaseConnection* لإمكانية استخدامه مع أي برنامج وأي نظام آخر دون تحديد اسمه.

تعديل الأسماء يكون صعباً في حال أنه واجهة برمجية interface مثل اسم خدمة ويب أو إجراء ضمنها أو حتى بارامتر وذلك لأنه اتفاق وعقد يجب عدم الإخلال به مع برامج أخرى ومع جهات أخرى تستخدم هذه الأسماء لتنفيذ تلك الدوال فإذا غيرنا في هذه الأسماء فإن البرامج المستفيدة من هذه الخدمات سوف تتوقف. كذلك عند كتابة مكتبة تُستخدم في أكثر من نظام أو منشورة في النت، فإن تغيير اسمها أو بصمة الدالة (كما تُسمى) يجعل البرامج المعتمدة عليها ينتج عنها خطأ في حالة إعادة ترجمتها.
من الأهداف التي نحققها باستخدام طريقة صحيحة للتسمية هي:

- سهولة قراءة وصيانة الكود
- تحقيق التجريد
- تحقيق وتسهيل إعادة الاستخدام
- سهولة توقع وظيفة أي جزء من الكود
- معرفة نقاط الضعف في أجزاء معينة من الكود مثلاً الارتباط

المحافظة على المبادئ الصحية

من التحديات الكبيرة في تطوير البرامج هو المحافظة على أن يكون التصميم صحيحاً، حيث أن الاستمرار في تطوير البرامج وإضافة طلبات جديدة من الزبائن والتغييرات المستمرة ودخول عدد من المبرمجين في دورة تطوير ذلك النظام ربما ينتج عنه انحراف في التصميم المتفق عليه، ويمكن أن يكتُب أحد المبرمجين كود بطريقة غير صحيحة مثل التكرار وعدم التجريد في الإجراءات والفئات وغيرها. فإذا زادت الأخطاء التصميمية فإن ذلك النظام يتحول من نظام ذو تصميم صحيح إلى نظام ذو تصميم غير صحيح وتبدأ تظهر عليه علامات ودلالات التصميم السيئ التي تكلمنا عنها سابقاً، وربما ينتهي به المطاف لفشله.
يمكن المحافظة على التصميم الجيد للنظام بمراقبة ومراجعة الإضافات الجديدة، وفي حال أن هناك إضافة أو تعديل حدث بطريقة غير صحيحة يمكن عمل إعادة صياغة لهذه الجزئية Refactoring وكلما كان هذا التصحيح في مرحلة مبكرة، كان أسهل. لكن عند التوغل في الأخطاء يصعب حينها التصحيح ويصبح مهمة مكلفة.

إعادة صياغة الكود Code Refactoring

الكود المكتوب بطريقة غير صحيحة أو الذي كان يتبع النهج الصحيح ثم بعد التغييرات الكثيرة أصبحت بعض أجزائه فيها إخلال بهذه القواعد فإنه يمكن إصلاحه بعملية تُسمى إعادة الصياغة refactoring، وإعادة الصياغة هي عملية تعديل الكود وتصحيحه من تلك المشاكل ليرجع إلى قواعد هندسة البرمجيات الصحيحة، لكن دون إضافة ميزات جديدة أو تحسين في الأداء مثلاً. فيكون الناتج لتشغيل البرنامج هو نفسه بدون اختلاف، ما اختلف هو التنظيم الداخلي. أما إذا حدث تغيير في الأداء مثلاً أو إضافة ميزات جديدة فهذا يُعد تعديل للبرنامج أو إضافة أو تحسين للأداء، لكن لا يمنع أن يسبقها إعادة صياغة قبل التعديل.

في الأمثلة السابقة، إبتداءً من مثال التجريد والذي كان به إجراء به كود خاص بملف معين، فقد أعدنا صياغته ليصبح إجراء عام، فهذه كانت أول عملية إعادة صياغة استخدمناها لمعالجة إجراء مكتوب بطريقة غير صحيحة، لكن الناتج في تنفيذ البرنامج لم يتغير سواءً مكتوب بطريقة صحيحة أم لا. كذلك باقي الأمثلة في مبدأ الوظيفة الواحدة و مبدأ الغلق والفتح في المثال الثاني الذي استخدمنا فيه تعدد الأشكال.

يحتاج من يقوم بهذه العملية (إعادة الصياغة) أن يكون ملّماً بالمبادئ الهندسية الصحيحة للبرمجة حتى يصحح كود كتبته بنفسه أو كود كتبه شخص آخر.

عملية إعادة الصياغة تسمح بالاستمرار في تطوير البرنامج وتسهيل صيانته واكتشاف العثرات. أحياناً عند عدم معرفة مشكلة ما بسبب أن الكود مكتوب بطريقة غير مقروئة يلجأ المطور إلى عملية إعادة الصياغة والتحسين لتقسيم البرنامج والإجراءات إلى أجزاء أصغر يسهل متابعتها واختبارها ثم اكتشاف المشاكل، كذلك يسمح بعمل إضافات جديدة بسهولة. ومن الأخطاء الكبيرة التي تحدث هو البناء على كود مكتوبة بطريقة غير صحيحة، مثلاً كتابة ميزات جديدة وكتابة كود جديد بنفس النسق غير الصحيح فيزيد التعقيد وتُصبح مهمة علاج الكود أصعب في المستقبل أصعب، فيلجأ مدير المشروع إلى حل آخر هو إعادة كتابة الكود من جديد، ويمكن في هذه الحالة استخدام لغة برمجة مختلفة.

عملية إعادة صياغة الكود لابد أن تُستخدم فيها نفس لغات البرمجة، وعلى نفس قاعدة الكود القديم، فإذا تغيرت لغة البرمجة المستخدمة أو بدأ المبرمجون كتابة إجزاء برمجية من الصفر، بما يُسمى بالتصميم النظيف (clean design) أصبح إسمها إعادة هندسة Re-engineering.

توجد مشكلة إدارية في إعادة صياغة الكود، حيث أن مدير المشروع لا يستطيع إقناع الزبون بالوقت المطلوب او التكلفة لهذه العملية، لأنه لا يوجد نتيجة ملموسة لها في تشغيل البرنامج، فلو أن الفريق البرمجي مكث شهراً لإعادة الصياغة لما كان هناك أي تغيير في واجهة المستخدم أو أداء النظام، لذلك من الأفضل إدخال إعادة الصياغة في دورة تطوير البرنامج وإضافة ميزات له، بحيث تكون إعادة الصياغة أولاً قبل بداية التعديل أو الإضافة، ثم تأتي عملية الإضافة والتعديل على كود أكثر وضوحاً وأكثر صحة، و من ناحية عملية يمكن عمل إعادة صياغة للأجزاء المراد تعديلها فقط دون المساس بالأجزاء المستقرة من الكود إلى أن يأتي وقت نحتاج فيه للتعديل في تلك الأجزاء المستقرة.

إعادة هندسة الكود Re-engineering

إعادة الهندسة المقصود بها إعادة كتابة البرنامج أو الوحدة من الصفر، بطريقة مختلفة و نظيفة clean design، أو بلغة برمجة مختلفة، والسبب إما أن كود البرنامج الأول به أخطاء تصميمية وأصبح معقد لدرجة لا تصلح معها إعادة الصياغة، أو أن إعادة الصياغة مكلفة فالأفضل منها كتابة البرنامج من الصفر. السبب الثاني أن هُناك مشكلة بالأداء في لغة البرمجة المستخدمة، فيكون الهدف إعادة الهندسة لاستخدام لغة برمجة ومترجم آخر يختلف بطريقة جذرية عن اللغة الأولى المستخدمة والتي أصبحت غير مناسبة لتحقيق الأداء المطلوب من الوحدة البرمجية المقصودة.
هدف آخر هو سلوك طريق جديد وأبسط في كتابة الكود، مثلاً استخدام خوارزميات مختصرة، أو ربط البرامج بواسطة بروتوكول مختلف. كذلك يمكن أن يكون الهدف هو لغة برمجة جديدة بها إمكانات أفضل ومزايا غير موجودة في اللغة القديمة، وذلك لأن البرامج يستمر تطويرها لأعوام طويلة فتختلف فيها اللغة الأفضل مع مرور الوقت، مثلاً لغة البرمجة التي كانت أفضل لمثل هذه البرامج في عام 2010 يمكن أن لا تكون هي نفسها اللغة الأفضل لهذه البرامج في عام 2022.

يمكن إعادة هندسة جزئية برمجية بعينها في نظام ما، دون إعادة هندسة النظام كاملاً خصوصاً إذا كان النظام مقسم إلى أقسام منفصلة عن بعضها لتحقيق مبدأ الوظيفة الواحدة، فمثل هذه الأنظمة مناسبة لإعادة كتابة بعض اجزائها بلغات برمجة أفضل تمثل الوقت الحالي، بخلاف لغة البرمجة الأولى التي كانت تمثل وقتاً مضى.

المراجع

1. **The Principles of Good Programming**
by by Christopher Diggins July 24, 2011
https://www.artima.com/weblogs/viewpost.jsp?thread=331531

2. **Wikipedia**
https://en.wikipedia.org/wiki/Coupling_(computer_programming)
https://en.wikipedia.org/wiki/Information_hiding
https://en.wikipedia.org/wiki/Cohesion_(computer_science)

الخاتمة

في الختام نتمنى أن يُستفاد من مادة هذا الكتاب وأن يكون عوناً على المبرمجين والمطورين والمعماريين والمصممين لتطوير برامجهم بطريقة سليمة تحقق كل أهداف هندسة البرمجيات وتجعل برامجنا تنافس البرامج العالمية من حيث الجودة والاعتمادية، بل وتنافسها في الكفاءة الإنتاجية لصناعة البرمجيات.

سبحانك اللهم وبحمدك، أشهد أن لا إله إلا أنت، أستغفرك وأتوب إليك ـ